SOFTWARE DEFINED TRANSPORT

BY

CHI-YAO HONG

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Doctoral Committee:

Assistant Professor P. Brighten Godfrey, Chair and Co-director of Research
Assistant Professor Matthew Caesar, Director of Research
Professor Klara Nahrstedt
Professor Indranil Gupta
Professor Nick Feamster, Georgia Institute of Technology

ABSTRACT

We advocate a *software defined transport* (SDT) architecture in which a transport controller schedules and dynamically re-schedules the flow sending rates based on current network conditions and the network operator's goals. This dissertation shows that this architecture provides both high *flexibility*, by allowing the operator to implement new transport policies as needed, and *fine-grained flow control*, by optimizing network resource allocation at flow-level in real time.

We begin with proposing a fine-grained flow scheduling protocol to complete flows quickly and meet flow deadlines. Through extensive packet-level and flow-level simulation, we demonstrate that fine-grained flow control can significantly reduce mean flow completion times by 30% or more compared with TCP, RCP, and D^3. We next design a software-driven controller which centrally allocates network resource such as bandwidth and routing paths for flexibility. In particular, we develop a prototype of our design for inter-datacenter wide area networks to achieve nearly optimal network utilization and service-level fairness. After that, we address network update problem to ensure bandwidth requirements during network updates subject to network capacity and switch memory constraints. Finally, we design and implement a fast, fine-grained flow-rate controller for data center networks. We show this design provides high scalability, by rate-controlling 95% of bytes of a cluster with several thousand servers within hundreds of milliseconds with a multi-threaded resource allocation algorithm, and application-level improvement, by reducing average shuffling times of MapReduce workload by $12-20\%$.

To my family and my friends.

ACKNOWLEDGMENTS

I am immensely grateful to my thesis advisor, Matthew Caesar. He has provided me all kinds of help throughout my PhD studies. When I entered this department as a foreign student who barely spoke English and struggled with research, I would not survive here without his tremendous help. He really cares about my success and had taken so much time to help me succeed in my research.

I am also very fortunate to be advised by Brighten Godfrey. He is my role model. I am lucky to have had the privilege to learn a ton of things from him by conducting research with him as a research assistant, learning teaching and presentation skills as a student in his class and as his teaching assistant.

I am honored to collaborate with many amazing people. Much of this dissertation is joint work with Ratul Mahajan, Srikanth Kandula, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. I have been lucky to collaborate with Jia Wang and Nick Duffield when interning at AT&T Labs Research. I also thank my MSR mentors Fang Yu and Yinglian Xie for teaching me how to conduct data analysis through distributed computing. Many thanks to my collaborators Nikita Borisov, Chia-Chi Lin, Pi-Cheng Hsiu, Tony Huang, Prateek Mittal, Shishir Nagaraja, Giang Nguyen, Ai-Chun Pang, Lucian Popa. I especially thank Ankit Singla for the invaluable friendship and help.

Nick Feamster, Indranil Gupta, and Klara Nahrstedt served on my thesis committee and provided useful comments for this dissertation. Jennifer Rexford gave invaluable feedback on my work at the early stages, and I also appreciate her generous help in my job search.

I feel deeply indebted to my lab colleagues: Rachit Agarwal, Jason Croft, Mo Dong, Soudeh Ghorbani, Kyle Jao, Qingxi Li, Virajith Jalaparti, Sangeetha Abdu Jyothi, Ahmed Khurshid, Oliver Michel, Rashid Tahir, Ashish Vulimiri, Anduo Wang, Fan Yang, Wenxuan Zhou, Kelvin Zou. I thank the feedback and the comments they gave on my work.

I thank my friends for giving me insightful comments: Saurabh Agarwal, Ahsan Arefin, Kai-Wei Chang, Ming-Wei Chang, Qieyun Dai, Wei Dong, Faraz Faghri, Yunchao Gong, I-Hong Hou, Shih-Wen Huang, Kevin Jin, Fariba Khan, Luke Kung, Lue-Jane Lee, Shen Li, Haohui Mai, Mirko Montanari, Chunyi Peng, Shu Shi, Lu Su, David Tan, Abhishek Verma, Li-Lun Wang, Po-Liang Wu, Xin Wu, and James Zeng. They also enriched my daily life and balanced my life from work.

This work is funded by NSF, Cisco, Symantec, and DARPA. I would like to thank those who provided help on this dissertation but did not get mentioned here. Please forgive me for not properly giving credit to you. I truly appreciate your support and contribution to my work.

I appreciate my family members for their love and support throughput my PhD studies. Especially thank my parents Yue-Ying Lin and Sheng-Wen Hong for their everlasting love.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Cloud infrastructures are increasingly expected to provide vital support for modern, data-intensive applications such as data-intensive distributed computing (e.g., MapReduce [3], Dryad [4], Scope [5] and FlumeJava [6]), large graph/matrix computation (e.g., Pregel [7] and Giraph [8]), and online services (e.g., web search, social networking and recommendation systems). The large variety of cloud applications have demanded a diverse range of service requirements such as optimizing completion times [9–12], meeting task deadlines [10,13], and satisfying fairness constraints across tenants [14–16]. However, legacy transport protocols used today, such as TCP, are known to be ill-suited for meeting modern application requirements [10, 13].

To satisfy the service requirements for increasingly emerged cloud applications, this dissertation argues that cloud network infrastructure needs two critical properties: (a) *flexibility*: Enabling desirable transport policies (e.g., prioritizing flows, satisfying flow deadlines, providing tenant-level bandwidth guarantees [17], and supporting high interactivity for video conferencing [18]) with little overhead cost, and (b) *fine-grained flow control*: Scheduling network resource at flow-level to optimize network performance in real time. Although recent work has focused on providing fine-grained flow control (e.g., D^3 [13], D2TCP [11], DeTail [12], pFabric [9]), they have limited flexibility, as each has its own subset of transport policies that it supports, and most of them requires custom modifications to switches. Consequently, these protocols have seen little practical use, arguably due to their low flexibility. For example, XCP (2002) [19] and RCP (2006) [20], require only a few CPU cycles per packet at switches and end-hosts, have not been deployed in today's Ethernet switches.

If we can achieve flexibility and fine-grained flow control, the network operator can adapt to use more efficient transport policies that help optimize the network performance to meet service requirements and greatly reduce the networking cost. For example, with better transport policies to ensure that

important traffic can always be protected, we can greatly save network cost by using a less aggressive capacity provisioning scheme, i.e., aggressively multiplexing high business priority traffic with low-priority maintenance traffic that can be delayed.

We summarize the research challenges when approaching these two properties as follows.

Flexibility: Although a software-based design provides better flexibility, it remains unclear what benefits it provides, what service requirements are desirable, and how to design proper resource allocation frameworks that support these service requirements. For example, if we update a network in a centralized fashion simplistically, the configurations can also cause severe, transient congestion because different senders and switches may apply the configurations at different times.

Fine-grained flow control: Given the sub-second timescales needed by flow-rate control, centralized scheduling of thousands of flows would raise a serious latency and scalability concern. How far can we push a flexible transport architecture towards real-time, fine-grained flow-rate control? In particular, we need scalable mechanisms for resource scheduling that maximizes application utilities subject to service constraints such as fairness.

The key contribution of this dissertation includes:

Chapter 2 presents a distributed transport rate-control protocol to achieve fine-grained flow control. By enabling flow preemption, this protocol can approximate several scheduling disciplines, such as a shortest job first algorithm to give priority to the short flows by pausing the contending flows. With this design, we demonstrate strong benefits of having fine-grained flow control over existing datacenter transport mechanisms.

Chapter 3 proposes a *Software Defined Transport* architecture (SDT), where a central transport controller schedules and dynamically re-schedules the flow sending rates and network forwarding plane. We demonstrate that this design significantly boosts inter-datacenter network utilization by centrally controlling when and how much traffic each service sends and frequently reconfiguring the network's data plane to match current traffic demand.

Chapter 4 studies how to ensure bandwidth requirements during network update in SDT. In particular, we study how to achieve congestion-free network update with practical constraints such as network update time, link capacity

and switch memory constraints.

Chapter 5 studies SDT's scalability limits and application-level improvement in datacenter networks. We propose a fast, multi-threaded rate allocation algorithm that emulates priority and fair queueing, and we found such design provides high scalability, by scheduling flow-rates of 95% of bytes in a cluster with several thousand servers within a few hundreds of milliseconds, and better application-level performance, by saving the average shuffling times by $12-20\%$ for MapReduce workload.

We briefly discuss each part of this dissertation next.

1.1 Finishing flows quickly with preemptive scheduling

We first present Preemptive Distributed Quick (PDQ) flow scheduling, a protocol designed to complete flows quickly and meet flow deadlines. PDQ borrows ideas from centralized scheduling disciplines and implements them in a fully distributed manner, making it scalable to today's data centers. Although this design requires switch modification, it demonstrates clear benefits of fine-grained flow scheduling in cloud infrastructures.

PDQ builds on traditional real-time scheduling techniques: when processing a queue of tasks, scheduling in order of Earliest Deadline First (EDF) is known to minimize the number of late tasks, while Shortest Job First (SJF) minimizes mean flow completion time. To perform dynamic decentralized scheduling, PDQ provides a distributed algorithm to allow a set of switches to collaboratively gather information about flow workloads and converge to a stable agreement on allocation decisions.

Unlike "fair sharing" protocols, EDF and SJF rely on the ability to *preempt* existing tasks, to ensure a newly arriving task with a smaller deadline can be completed before a currently-scheduled task. To support this functionality in distributed environments, PDQ provides the ability to perform distributed preemption of existing flow traffic, in a manner that enables fast switchover and is guaranteed to never deadlock.

We demonstrate that PDQ can save $\sim$30% average flow completion time compared with TCP, RCP and D^3 [13]; and can support $3\times$ as many concurrent

senders as D^3 while meeting flow deadlines. Moreover, we show that PDQ is stable, resilient to packet loss, and preserves nearly all its performance gains even given inaccurate flow information. We develop and evaluate a multipath version of PDQ, showing further performance and reliability gains.

1.2 Achieving high utilization in inter-datacenter WANs

We propose SWAN, a SDT-based resource manager in the context of inter-datacenter WAN for optimizing network utilization. It centrally controls when and how much traffic each service sends and frequently re-configure the network's data plane to match current traffic demand. Compared with MPLS TE, the current practice used in many WANs, SWAN provides better efficiency by globally coordinating service transport rates and paths. By explicitly controlling service sending rates, SWAN can easily achieve fairness across services.

While this architecture is conceptually simple, we need scalable designs to control network flows in global inter-datacenter networks. To this end, we present a hierarchical control architecture and use aggregated service contracts for scalability. We also develop a scalable resource allocation algorithm that meet the desirable service policies and constraints, including preferential treatment for higher-priority services and fairness among similar services.

We implement a prototype of SWAN and demonstrate its feasibility via a series of experiments using production inter-DC traffic. We found SWAN provides high efficiency, carrying 60% more traffic than the current practice, and fairness, approximating max-min fair share rate well.

1.3 Congestion-free update

Although SDT's concept is fairly simple, frequent network re-configurations can also cause severe, transient congestion because different devices may apply updates at different times. We develop a novel technique that leverages a small amount of scratch capacity on links to apply updates in a provably congestion-free manner, without making any assumptions about the order and timing of updates at individual switches.

To scale to large networks in the face of limited forwarding table capacity, we greedily selects a small set of entries that can best satisfy current demand. It updates this set without disrupting traffic by leveraging a small amount of scratch capacity in forwarding tables.

Experiments using a testbed prototype and data-driven simulations of two production networks show that this design can ensure worst-case network over-subscription ratio and provides a much better experience during network reconfiguration.

1.4 Scalability limits and application improvement

SDT lies in the family of *fabric* architectures [21] which use central control to send instructions to edge devices (in our case, end-hosts), allowing the core of the network to provide only basic packet forwarding functions. But the fabric architecture seems infeasible for fine-grained flow-rate control: given the sub-second timescales needed by rate control, centralized scheduling of thousands of servers' flows would raise a serious latency and scalability concern. Given the sub-second timescales needed by rate control, centralized scheduling of thousands of flows would raise a serious latency and scalability concern. How far can we push this flexible transport architecture towards real-time, fine-grained rate control?

To tackle with this latency issue, we use two techniques. First, we propose a multi-threaded rate allocation algorithm that emulates link-level queueing disciplines. We found such a multi-threaded design greatly reduces computational time by allowing parallel processing of several links. This scalable design provides several useful transport scheduling disciplines such as prioritization and fair queueing. Second, we handle short, transient flows without the central controller. Based upon proper packet header marking (e.g., via DSCP bits) at end-hosts, short flows are initiated with highest queueing priority and do not need to be scheduled by the transport controller until they send more than a certain number of bytes.

We have implemented these two techniques in a testbed prototype, and the experimental results show that our design can scale this architecture reasonably well—a single central server can schedule and dynamically re-schedule flow-rates of 95% of bytes in a cluster with several thousand servers within

hundreds of milliseconds. Because these 95% of bytes are composed of flows with duration longer than 10 seconds [22], scheduling these flows with a much smaller control interval of hundreds of milliseconds is feasible.

Although our design clearly improves network performance (e.g., achieving high utilization, minimizing flow completion times, and meeting flow deadlines), it's unclear how much application-level improvement it will bring up. As a first step, we extend this design to handle MapReduce workloads by prioritizing jobs and demonstrate that our design can save average shuffling times by $12 - 20\%$.

1.5 Thesis roadmap

Chapter 2 presents the benefits of fine-grained flow scheduling. Chapter 3 shows how SDT helps boost network utilization in inter-datacenter WANs. Chapter 4 studies how to ensure bandwidth and switch memory constraints during network update. Chapter 5 studies scalability limits and performance improvement on applications. Chapter 6 concludes with open problems.

CHAPTER 2

PREEMPTIVE DISTRIBUTED FLOW SCHEDULING

2.1 Background

Data centers are now used as the underlying infrastructure of many modern commercial operations, including web services, cloud computing, and some of the world's largest databases and storage services. Data center applications including financial services, social networking, recommendation systems, and web search often have very demanding latency requirements. For example, even fractions of a second make a quantifiable difference in user experience for web services [23]. And a service that aggregates results from many back-end servers has even more stringent requirements on completion time of the back-end flows, since the service must often wait for the *last* of these flows to finish or else reduce the quality of the final results.[1] Minimizing delays from network congestion, or meeting soft-real-time deadlines with high probability, is therefore important.

Unfortunately, current transport protocols neither minimize flow completion time nor meet deadlines. TCP, RCP [20], ICTCP [25], and DCTCP [24] approximate *fair sharing*, dividing link bandwidth equally among flows. Fair sharing is known to be far from optimal in terms of minimizing flow completion time [26] and the number of deadline-missing flows [27]. As a result, a study of three production data centers [13] showed that a significant fraction ($7 - 25\%$) of flow deadlines were missed, resulting in degradation of application response quality, waste of network bandwidth, and ultimately loss of operator revenue [24].

This chapter presents **Preemptive Distributed Quick (PDQ)** flow scheduling, a protocol designed to complete flows quickly and meet flow deadlines. PDQ builds on traditional real-time scheduling techniques: when processing a

[1] See discussion in [24], §2.1.

queue of tasks, scheduling in order of Earliest Deadline First (EDF) is known to minimize the number of late tasks, while Shortest Job First (SJF) minimizes mean flow completion time. However, applying these policies to scheduling data center flows introduces several new challenges.

First, EDF and SJF assume a centralized scheduler which knows the global state of the system; this would impede our goal of low latency in a large data center. To perform dynamic decentralized scheduling, PDQ provides a distributed algorithm to allow a set of switches to collaboratively gather information about flow workloads and converge to a stable agreement on allocation decisions. Second, unlike "fair sharing" protocols, EDF and SJF rely on the ability to *preempt* existing tasks, to ensure a newly arriving task with a smaller deadline can be completed before a currently-scheduled task. To support this functionality in distributed environments, PDQ provides the ability to perform distributed preemption of existing flow traffic, in a manner that enables fast switchover and is guaranteed to never deadlock.

Thus, PDQ provides a distributed flow scheduling layer which is *lightweight*, using only FIFO tail-drop queues, and *flexible*, in that it can approximate a range of scheduling disciplines based on relative priority of flows. We use this primitive to implement two scheduling disciplines: EDF to minimize mean flow completion time, and SJF to minimize the number of deadline-missing flows.

Through an extensive simulation study using real datacenter workloads, we find that PDQ provides strong benefits over existing datacenter transport mechanisms. PDQ is most closely related to D^3 [13], which also tries to meet flow deadlines. Unlike D^3, which is a "first-come first-reserve" algorithm, PDQ proactively and preemptively gives network resources to the most critical flows. For deadline-constrained flows, our evaluation shows PDQ supports 3 times more concurrent senders than [13] while satisfying their flow deadlines. When flows have no deadlines, we show PDQ can reduce mean flow completion times by ∼30% or more compared with TCP, RCP, and D^3.

The key contributions of this chapter are:

- We design and implement PDQ, a distributed flow scheduling layer for data centers which can approximate a range of scheduling disciplines.
- We build on PDQ to implement flow scheduling disciplines that minimize mean flow completion time and the number of deadline-missing flows.

- We demonstrate PDQ can save ∼30% average flow completion time compared with TCP, RCP and D^3; and can support 3× as many concurrent senders as D^3 while meeting flow deadlines.
- We show that PDQ is stable, resilient to packet loss, and preserves nearly all its performance gains even given inaccurate flow information.
- We develop and evaluate a multipath version of PDQ, showing further performance and reliability gains.

2.2 Overview

We start by presenting an example to demonstrate potential benefits of PDQ over existing approaches (§2.2.1). We then give a description of key challenges that PDQ must address (§2.2.2).

2.2.1 Example of Benefits

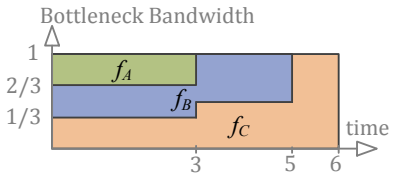
Example of benefits. Consider the scenario shown in Figure 2.1, where three concurrent flows (f_A, f_B, and f_C) arrive simultaneously.

Deadline-unconstrained Case: Suppose that the flows have no deadlines, and our objective is to minimize the average flow completion time. Assuming a fluid traffic model (infinitesimal units of transmission), the result given by fair sharing is shown in Figure 2.1b: [f_A,f_B,f_C] finish at time [3,5,6], and the average flow completion time is $\frac{3+5+6}{3} = 4.67$. If we schedule the flows by SJF (one by one according to flow size), as shown in Figure 2.1c, the completion time becomes $\frac{1+3+6}{3} = 3.33$, a savings of ∼29% compared to fair sharing. Moreover, for every individual flow, the flow completion time in SJF is no larger than that given by fair sharing.

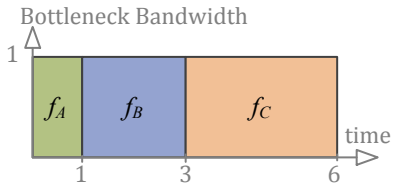
Deadline-constrained Case: Suppose now that the flows have deadlines, as specified in Figure 2.1a. The objective becomes minimizing the number of tardy flows, i.e., maximizing the number of flows that meet their deadlines. For fair sharing, both flow f_A and f_B fail to meet their deadlines, as shown in Figure 2.1b. If we schedule the flows by EDF (one by one according to deadline), as shown in Figure 2.1c, every flow can finish before its deadline.

Flow ID	Size	Deadline
f_A	1	1
f_B	2	4
f_C	3	6

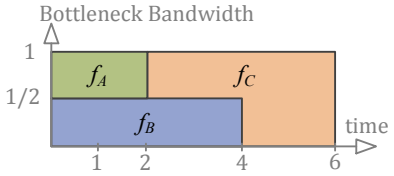
(a)



(b)



(c)



(d)

Figure 2.1: Motivating Example. (a) Three concurrent flows competing for a single bottleneck link; (b) Fair sharing; (c) SJF/EDF; (d) D^3 for flow arrival order $f_B \rightsquigarrow f_A \rightsquigarrow f_C$.

Now we consider D^3, a recently proposed deadline-aware protocol for data center networks [13]. When the network is congested, D^3 satisfies as many flows as possible according to the flow request rate in the order of their arrival. In particular, each flow will request a rate $r = \frac{s}{d}$, where s is the flow's size and d is the time until its deadline. Therefore, the result of D^3 depends highly on flow arrival order. Assuming flows arrive in the order $f_B \rightsquigarrow f_A \rightsquigarrow f_C$, the result of D^3 is shown in Figure 2.1d. Flow f_B will send with rate $\frac{2}{4} = 0.5$ and will finish right before its deadline. However, flow f_A, which arrives later than f_B, will fail to meet its deadline using the remaining bandwidth, as evident in Figure 2.1d. In fact, out of $3! = 6$ possible permutations of flow arrival order, D^3 will fail to satisfy some of the deadlines for 5 cases, the only exception being the order $f_A \rightsquigarrow f_B \rightsquigarrow f_C$, which is the order EDF chooses. Although D^3 also allows senders to terminate flows that fail to meet their deadlines (to save bandwidth), termination does not help in this scenario and is not presented in Figure 2.1d.

2.2.2 Design Challenges

Although attractive performance gains are seen from the example, many design challenges remain to realize the expected benefits.

Decentralizing Scheduling Disciplines: Scheduling disciplines like EDF or

SJF are centralized algorithms that require global knowledge of flow information, introducing a single point of failure and significant overhead for senders to interact with the centralized coordinator. For example, a centralized scheduler introduces considerable flow initialization overhead, while becoming a congestive hot-spot. This problem is especially severe in data center workloads where the majority of flows are mice. A scheduler maintaining only elephant flows like DevoFlow [28] seems unlikely to succeed in congestion control as deadline constraints are usually associated with mice. The need to address the above limitations leads to PDQ, a fully distributed solution where switches collaboratively control flow schedules.

Switching Between Flows Seamlessly: The example of §2.2.1 idealistically assumed we can start a new flow immediately after a previous one terminates, enabling all the transmission schedules (Figure 2.1b, 2.1c and 2.1d) to fully utilize the bottleneck bandwidth and thus complete flows as quickly as possible. However, achieving high utilization during flow switching in practice requires precise control of flow transmission time. One could simplify this problem by assuming synchronized time among both switches and senders, but that introduces additional cost and effort to coordinate clocks. PDQ addresses this problem by starting the next set of waiting flows slightly before the current sending flows finish.

Prioritizing Flows using FIFO Tail-drop Queues: One could implement priority queues in switches to approximate flow scheduling by enforcing packet priority. Ideally, this requires that each of the concurrent flows has a unique priority class. However, a data center switch can have several thousand active flows within a one second bin [2], while modern switches support only ∼10 priority classes [13]. Therefore, for today's data center switches, the number of priority classes per port is far below the requirements of such an approach, and it is unclear whether modifying switches to support a larger number of priority classes can be cost-effective. To solve this, PDQ explicitly controls the flow sending rate to regulate flow traffic and retain packets from low-priority flows at senders. With this flow pausing strategy, PDQ only requires per-link FIFO tail-drop queues at switches.

2.3 Protocol

Centralized Algorithm: To clarify our approach, we start by presenting it as an idealized *centralized* scheduler with complete visibility of the network, able to communicate with devices in the network with *zero delay*. To simplify exposition, the centralized scheduler assumes that flows have no deadlines, and our only goal is to optimize flow completion time. We will later relax these assumptions.

We define the *expected flow transmission time*, denoted by $\mathscr{T}_i$ for any flow i, to be the remaining flow size divided by its *maximal sending rate* $\mathscr{R}_i^{\max}$. The maximal sending rate $\mathscr{R}_i^{\max}$ is the minimum of the sender NIC rate, the switch link rates, and the rate that receiver can process and receive. Whenever network workload changes (a new flow arrives, or an existing flow terminates), the centralized scheduler recomputes the flow transmission schedule as follows:

1. $B_e =$ available bandwidth of link e, initialized to e's line rate.
2. For each flow i, in increasing order of $\mathscr{T}_i$:
   (a) Let P_i be flow i's path.
   (b) Send flow i with rate $\mathscr{R}_i^{\text{sch}} = \min_{\forall e \in P_i}(\mathscr{R}_i^{\max}, B_e)$.
   (c) $B_e \leftarrow B_e - \mathscr{R}_i^{\text{sch}}$ for each $e \in P_i$.

Distributed Algorithm: We eliminate the unrealistic assumptions we made in the centralized algorithm to construct a fully distributed realization of our design. To distribute its operation, PDQ switches propagate flow information via explicit feedback in packet headers. PDQ senders maintain a set of flow-related variables such as flow sending rate and flow size and communicate the flow information to the intermediate switches via a scheduling header added to the transport layer of each data packet. When the feedback reaches the receiver, it is returned to the sender in an ACK packet. PDQ switches monitor the incoming traffic rate of each of their output queues and inform the sender to send data with a specific rate ($\mathscr{R}>0$) or to pause ($\mathscr{R}=0$) by annotating the scheduling header of data/ACK packets. When the feedback reaches the receiver, it is returned to the sender in an ACK packet. PDQ switches monitor the incoming traffic rate of each of their output queues and inform the sender

to send data with a specific rate ($\mathscr{R}>0$) or to pause ($\mathscr{R}=0$) by annotating the scheduling header of data/ACK packets. We present the details of this distributed realization in the following sections.

2.3.1 PDQ Sender

Like many transport protocols, a PDQ sender sends a SYN packet for flow initialization and a TERM packet for flow termination, and re-sends a packet after a timeout. The sender maintains standard data structures for reliable transmission, including estimated round-trip time and states (e.g., timer) for in-flight packets. The PDQ sender maintains several state variables: its current sending rate ($\mathscr{R}_S$, initialized to zero), the ID of the switch (if any) who has paused the flow ($\mathscr{P}_S$, initialized to ø), flow deadline ($\mathscr{D}_S$, which is optional), the expected flow transmission time ($\mathscr{T}_S$, initialized to the flow size divided by sender NIC rate), the inter-probing time ($\mathscr{I}_S$, initialized to ø), and the measured RTT (RTT_S, estimated by an exponential decay).

The sender sends packets with rate $\mathscr{R}_S$. If the rate is zero, the sender sends a *probe* packet every $\mathscr{I}_S$ RTTs to get rate information from the switches. A probe packet is a packet with a scheduling header but no data content.

On packet departure, the sender attaches a scheduling header to the packet, containing fields set based on the values of each of the sender's state variables above. $\mathscr{R}_H$ is always set to the maximal sending rate $\mathscr{R}_S^{\max}$, while the remaining fields in the scheduling header are set to its current maintained variables. Note that the subscript H refers to a field in the scheduling header; the subscript S refers to a variable maintained by the sender; the subscript i refers to a variable related to the ith flow in the switch's flow list.

Whenever an ACK packet arrives, the sender updates its flow sending rate based on the feedback: $\mathscr{T}_S$ is updated based on the remaining flow size, RTT_S is updated based on the packet arrival time, and the remaining variables are set to the fields in the scheduling header.

Early Termination: For deadline-constrained flows, when the incoming flow demand exceeds the network capacity, there might not exist a feasible schedule for all flows to meet their deadlines. In this case, it is desirable to discard a minimal number of flows while satisfying the deadline of the remaining flows. Unfortunately, minimizing the number of tardy flows in a dynamic setting is

an $\mathcal{NP}$-complete problem.[2]

Therefore, we use a simple heuristic, called *Early Termination*, to terminate a flow when it cannot meet its deadline. Here, the sender sends a TERM packet whenever *any* of the following conditions happen:

1. Deadline is past (Time $> \mathcal{D}_S$).
2. The remaining flow transmission time is larger than the time to deadline (Time $+ \mathcal{T}_S > \mathcal{D}_S$).
3. The flow is paused ($\mathcal{R}_S = 0$), and the time to deadline is smaller than an RTT (Time $+ RTT_S > \mathcal{D}_S$).

2.3.2 PDQ Receiver

A PDQ receiver copies the scheduling header from each data packet to its corresponding ACK. Moreover, to avoid the sender overrunning the receiver's buffer, the PDQ receiver reduces $\mathcal{R}_H$ if it exceeds the maximal rate that receiver can process and receive.

2.3.3 PDQ Switch

The high-level objective of a PDQ switch is to let the most *critical* flow complete as soon as possible. To this end, switches share a common flow comparator, which decides flow criticality, to approximate a range of scheduling disciplines. In this study, we implement two disciplines, EDF and SJF, while we give higher priority to EDF. In particular, we say a flow is more critical than another one if it has smaller deadline (emulating EDF to minimize the number of deadline-missing flows). When there is a tie or flows have no deadline, we break it by giving priority to the flow with smaller expected transmission time (emulating SJF to minimize mean flow completion time). If a tie remains, we break it by flow ID. If desired, the operator could easily override the comparator to approximate other scheduling disciplines.

[2]Consider a subproblem where a set of concurrent flows that share a bottleneck link all have the same deadline. This subproblem of minimizing the number of tardy flows is exactly the $\mathcal{NP}$-complete subset sum problem [29].

The flow controller performs Algorithm 1 and 3 whenever it receives a data packet and an ACK packet, respectively. The flow controller's objective is to *accept* or *pause* the flow. A flow is accepted if *all* switches along the path accept it. However, a flow is paused if *any* switch pauses it. This difference leads to the need for different actions:

Pausing: If a switch decides to pause a flow, it simply updates the "pauseby" field in the header ($\mathscr{P}_H$) to its ID. This is used to inform other switches and the sender that the flow should be paused. Whenever a switch notices that a flow is paused by another switch, it removes the flow information from its state. This can help the switch to decide whether it wants to accept other flows.

Acceptance: To reach consensus across switches, flow acceptance takes two phases: (i) in the forward path (from source to destination), the switch computes the available bandwidth based on flow criticality (Algorithm 2) and updates the rate and pauseby fields in the scheduling header; (ii) in the reverse path, if a switch sees an empty pauseby field in the header, it updates the *global* decision of acceptance to its state ($\mathscr{P}_i$ and $\mathscr{R}_i$).

Early Start: Given a set of flows that are not paused by other switches, the switch accepts flows according to their criticality until the link bandwidth is fully utilized and the remaining flows are paused. Although this ensures that the more critical flows can preempt other flows to fully utilize the link bandwidth, this can lead to *low link utilization when switching between flows*. To understand why, consider two flows, A and B, competing for a link's bandwidth. Assume that flow A is more critical than flow B. Therefore, flow A is accepted to occupy the entire link's bandwidth, while flow B is paused and sends only probe packets, e.g., one per its RTT. By the time flow A sends its last packet (TERM), the sender of flow B does not know it should start sending data because of the feedback loop delay. In fact, it could take one to two RTTs before flow B can start sending data. Although the RTT in data center networks is typically very small (e.g., $\sim$150 μs), the high-bandwidth short-flow nature makes this problem non-negligible. In the worst case where all the flows are short control messages (<10 KByte) that could finish in just one RTT, links could be idle more than half the time.

To solve this, we propose a simple concept, called *Early Start*, to provide seamless flow switching. The idea is to start the next set of flows slightly before the current sending flows finish. Given a set of flows that are not paused

by other switches, a PDQ switch classifies a currently sending flow as nearly completed if the flow will finish sending in K RTTs (i.e., $\mathscr{T}_i < K \times RTT_i$), for some small constant K. We let the switch additionally accept as many nearly-completed flows as possible according to their criticality and subject to the resource constraint: aggregated flow transmission time (in terms of its estimated RTT) of the accepted nearly-completed flows ($\sum_i \mathscr{T}_i/RTT_i$) is no larger than K.

The threshold K determines how early and how many flows will be considered as nearly-completed. Setting K to 0 will prevent concurrent flows completely, resulting in low link utilization. Setting K to a large number will result in congested links, increased queue sizes, and increased completion times of the most critical flows. Any value of K between 1 and 2 is reasonable, as the control loop delay is one RTT and the inter probing time is another RTT. In our current implementation we set $K = 2$ to maximize the link utilization, and we use the /ate controller to drain the queue.

Dampening: When a more critical flow arrives at a switch, PDQ will pause the current flow and switch to the new flow. However, bursts of flows that arrive concurrently are common in data center networks, and can potentially cause frequent flow switching, resulting in temporary instability in the switch state. To suppress this, we use dampening: after a switch has accepted a flow, it can only accept other paused flows after a given small period of time, as shown in Algorithm 1.

Suppressed Probing: One could let a paused sender send one probe per RTT. However, this can introduce significant bandwidth overhead because of the small RTTs in data center networks. For example, assume a 1-Gbps network where flows have an RTT of 150 μs. A paused flow that sends a 40-byte probe packet per RTT consumes $\frac{40 \text{ Byte}}{150\ \mu s}/1$ Gbps $\approx 2.13\%$ of the total bandwidth. The problem becomes more severe with larger numbers of concurrent flows.

2.4 Formal properties

In this section, we present two formal properties of PDQ — deadlock-freedom and finite convergence time.

Assumptions: Without loss of generality, we assume there is no packet loss.

```
if 𝒫_H = other switch then
    Remove the flow from the list if it is in the list; return;
end
if the flow is not in the list then
    if the list is not full or the flow criticality is higher than the least
    critical flow in the list then
        Add the flow into the list with rate ℛ_i = 0. Remove the least
        critical flow from the list whenever the list has more than κ
        flows.
    end
    else
        Set ℛ_H to RCP fair share rate;
        if ℛ_H = 0 then 𝒫_H = myID;
        return;
    end
end
Let i be the flow index in the list; Update the flow information:
<𝒟_i,𝒯_i,RTT_i> = <𝒟_H,𝒯_H,RTT_H>;
if W=min(Availbw(i),ℛ_H)>0 then
    if the flow is not sending (𝒫_i ≠ ø), and the switch just accepted
    another non-sending flow then
        𝒫_H = myID; 𝒫_i = myID; // Pause it
    end
    else 𝒫_H = ø; ℛ_H=W; // Accept it
end
else 𝒫_H = myID; 𝒫_i = myID; // Pause it
```

Algorithm 1: PDQ Receiving Data Packet

```
X=0; A=0;
for (i = 0; i < j; i = i + 1) do
    if 𝒯_i/RTT_i < K and X < K then
        X = X + 𝒯_i/RTT_i;
    end
    else
        A = A + ℛ_i;
    end
    if A ≥ C then return 0;
end
return C − A;
```

Algorithm 2: Availbw(j)

Similarly, we assume flows will not be paused due to the use of flow dampening. Because PDQ flows periodically send probes, the properties we discuss in

1 **if** $\mathscr{P}_H = \textit{other switch}$ **then**
2 | Remove the flow from the list if it is in the list;
3 **end**
4 **if** $\mathscr{P}_H \neq \emptyset$ **then**
5 | $\mathscr{R}_H = 0$; // Flow is paused
6 **end**
7 **if** *the flow is in the list with index* i **then**
8 | $\mathscr{P}_i = \mathscr{P}_H$; $\mathscr{I}_H = \max\{\mathscr{I}_H, X \times i\}$; $\mathscr{R}_i = \mathscr{R}_H$;
9 **end**

Algorithm 3: PDQ Receiving ACK

this section will hold with additional latency when the above assumptions are violated. For simplicity, we also assume the link rate C is equal to the maximal sending rate $\mathscr{R}_S^{\max}$ (i.e., $\mathscr{R}_S^{\text{sch}} = 0$ or C). Thus, each link accepts only one flow at a time.

Definitions: We say a flow is *competing* with another flow if and only if they share at least one common link. Moreover, we say a flow F_1 is a *precedential* flow of flow F_2 if and only if they are competing with each other and flow F_1 is more critical than flow F_2. We say a flow F is a *driver* if and only if (i) flow F is more critical than any other competing flow, or (ii) all the competing flows of flow F that are more critical than flow F are non-drivers.

Results: In Appendix A, we verify that PDQ has no *deadlock*, which is a situation where two or more competing flows are paused and are each waiting for the other to finish (and therefore neither ever does). In Appendix B, we further prove that PDQ will converge to the *equilibrium* in $P_{\max} + 1$ RTTs for stable workloads, where $P_{\max}$ is the maximal number of precedential flows of any flow. Given a collection of active flows, the equilibrium is defined as a state where the drivers are accepted while the remaining flows are paused.

2.5 Performance

In this section, we evaluate PDQ's performance through comprehensive simulations. We first describe our evaluation setting (§2.5.1). Under a "query aggregation" scenario, PDQ achieves near-optimal performance and greatly outperforms D^3, RCP and TCP (§2.5.2). We then demonstrate that PDQ retains its performance gains under different workloads, including two realistic

data center workloads from measurement studies (§2.5.3), followed by two scenarios to demonstrate that PDQ does not compromise on traditional congestion control performance metrics (§2.5.4). Moreover, PDQ retains its performance benefits on a variety of data center topologies (Fat-Tree, BCube and Jellyfish) and provides clear performance benefits at all scales that we evaluated (§2.5.5). Further, we show that PDQ is highly resilient to inaccurate flow information and packet loss (§2.5.6).

2.5.1 Evaluation settings

Our evaluation considers two classes of flows:

Deadline-constrained Flows are time sensitive flows that have specific deadline requirements to meet. The flow size is drawn from the interval [2 KByte, 198 KByte] using a uniform distribution, as done in a prior study [13]. This represents query traffic (2 to 20 KByte in size) and delay sensitive short messages (>100 KByte) in data center networks [24]. The flow deadline is drawn from an exponential distribution with mean 20 ms, as suggested by [13]. However, some flows could have tiny deadlines that are unrealistic in real network applications. To address this, we impose a lower bound on deadlines, and we set it to 3 ms in our experiments. We use *Application Throughput*, the percentage of flows that meet their deadlines, as the performance metric of deadline-constrained flows. **Deadline-unconstrained Flows** are flows that have no specific deadlines, but it is desirable that they finish early. For example, Dryad jobs that move file partitions across machines. Similarly, we assume the flow size is drawn uniformly from an interval with a mean of 100/1000 KByte. We use the average flow completion time as the performance metric.

We have developed our own event-driven packet-level simulator written in C++. The simulator models the following schemes:

PDQ: We consider different variants of PDQ. We use PDQ(Full) to refer to the complete version of PDQ, including Early Start (ES), Early Termination (ET) and Suppressed Probing (SP). Likewise, we refer to the partial version of PDQ which excludes the above three algorithms as PDQ(Basic). To better understand the performance contribution of each algorithm, we further extend PDQ(Basic) to PDQ(ES) and PDQ(ES+ET). **D^3:** We implemented a complete version of D^3 [13], including the rate request processing procedure,

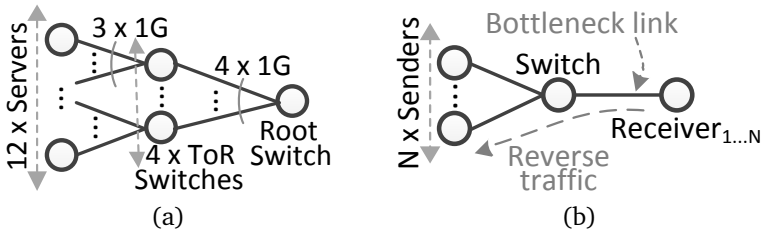


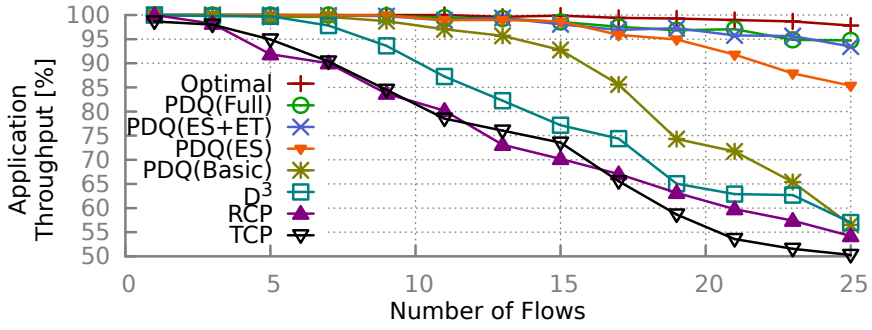
Figure 2.2: Example topologies: (a) a 17-node single-rooted tree topology; (b) a single-bottleneck topology: sending servers associated with different flows are connected via a single switch to the same receiving server. Both topologies use 1 Gbps links, a switch buffer of 4 MByte, and FIFO tail-drop queues. Per-hop transmission/propagation/processing delay is set to 11/0.1/25 μs.

the rate adaptation algorithm (with the suggested parameters $\alpha = 0.1$ and $\beta = 1$), and the quenching algorithm. In the original algorithm when the total demand exceeds the switch capacity, the fair share rate becomes negative. We found this can cause a flow to return the allocated bandwidth it already reserved, resulting in unnecessarily missed deadlines. Therefore, we add a constraint to enforce the fair share bandwidth fs to always be non-negative, which improves D^3's performance.

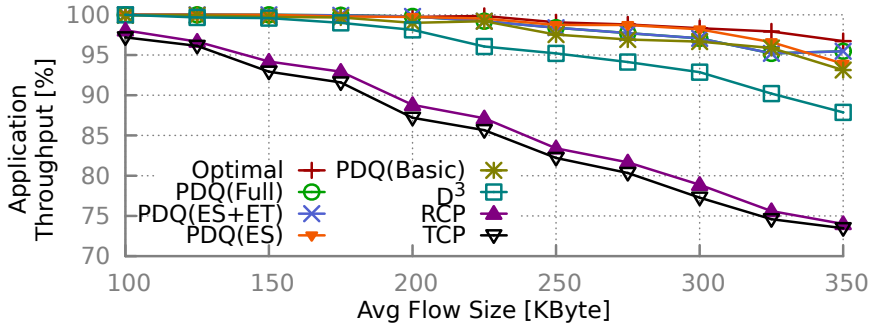
RCP: We implement RCP [20] and optimize it by counting the exact number of flows at switches. We found this improves the performance by converging to the fair share rate more quickly, significantly reducing the number of packet drops when encountering a sudden large influx of new flows [30]. This is exactly equivalent to D^3 when flows have no deadlines.

TCP: We implement TCP Reno and optimize it by setting a small RTO_{min} to alleviate the TCP Incast problem, as suggested by previous studies [24, 31].

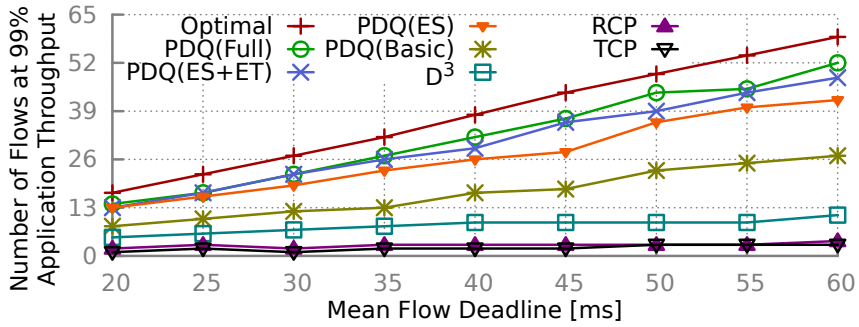
Unless otherwise stated, we use single-rooted tree, a commonly used data center topology for evaluating transport protocols [13, 24, 25, 31]. In particular, our default topology is a two-level 12-server single-rooted tree topology with 1 Gbps link rate, the same as used in D^3. The default traffic pattern is called *query aggregation*: a number of senders initiate flows at the same time to the same receiver (the aggregator). This is a very common application scenario in data center networks and has been adopted by a number of previous works [13, 24, 25].



(a)



(b)

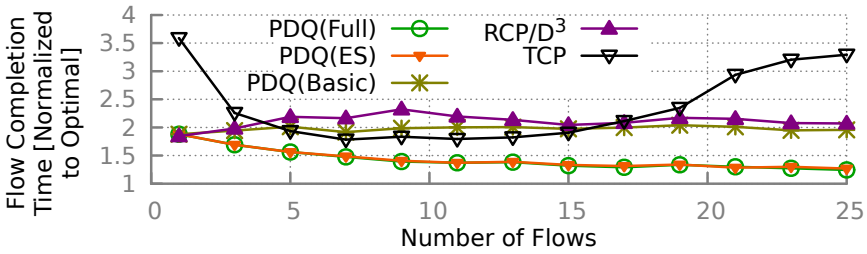


(c)

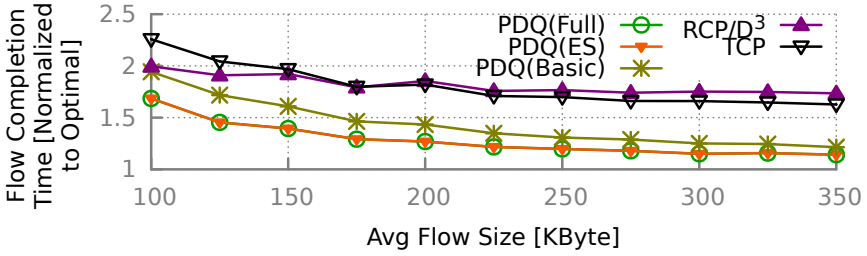
Figure 2.3: PDQ outperforms D^3, RCP and TCP and achieves near-optimal performance. Deadline-constrained flows.

2.5.2 Query Aggregation

In this section, we consider a scenario called *query aggregation*: a number of senders initiate flows at the same time to the same receiver (the aggregator). This is a very common application scenario in data center networks and has been adopted by a number of previous works [13, 24, 25]. We first evaluate the protocols in the deadline-constrained case, followed by the deadline-unconstrained case.



(a)



(b)

Figure 2.4: PDQ outperforms D^3, RCP and TCP and achieves near-optimal performance. Deadline-unconstrained flows.

Impact of Number of Flows: We start by varying the number of flows.[3] To understand bounds on performance, we also simulate an *optimal* solution, where an omniscient scheduler can control the transmission of any flow with no delay. It first sorts the flows by EDF, and then uses a dynamic programming algorithm to discard the minimum number of flows that cannot meet their deadlines (Algorithm 3.3.1 in [32]). We observe that PDQ has near-optimal application throughput across a wide range of loads (Figure 2.3a).

Figure 2.3a demonstrates that Early Start is very effective for short flows. By contrast, PDQ(Basic) has much lower application throughput, especially during heavy system load because of the long down time between flow switching. Early Termination further improves performance by discarding flows that cannot meet their deadline. Moreover, Figure 2.3a demonstrates that, as the number of concurrent flows increases, the application throughput of D^3, RCP and TCP decreases significantly.

Impact of Flow Size: We fix the number of concurrent flows at 3 and study the impact of increased flow size on the application throughput. Figure 2.3b shows that as the flow size increases, the performance of deadline-agnostic

[3]We randomly assign f flows to n senders while ensuring each sender has either $\lfloor f/n \rfloor$ or $\lceil f/n \rceil$ flows.

schemes (TCP and RCP) degrades considerably, while PDQ remains very close to optimal regardless of the flow size. However, Early Start and Early Termination provide fewer benefits in this scenario because of the small number of flows.

Impact of Flow Deadline: Data center operators are particularly interested in the operating regime where the network can satisfy almost every flow deadline. To this end, we attempt to find, using a binary search procedure, the maximal number of flows a protocol can support while ensuring 99% application throughput. We also vary the flow deadline, which is drawn from an exponential distribution, to observe the system performance with regard to different flow deadlines with mean between 20 ms to 60 ms. Figure 2.3c demonstrates that, compared with D^3, PDQ can support >3 times more concurrent flows at 99% application throughput, and this ratio becomes larger as the mean flow deadline increases. Moreover, Figure 2.3c shows that Suppressed Probing becomes more useful as the number of concurrent flows increases.

Next we discuss the results for deadline-unconstrained flows.

Impact of Flow Number: For deadline-unconstrained case, we first measure the impact of the number of flows on the average flow completion time. Overall, Figure 2.4a demonstrates that PDQ can effectively approximate the optimal flow completion time. The largest gap between PDQ and optimal happens when there exists only one flow and is due to flow initialization latency. RCP has a similar performance for the single-flow case. However, its flow completion time becomes relatively large as the number of flows increases. TCP displays a large flow completion time when the number of flows is small due to the inefficiency of slow start. When the number of concurrent flows is large, TCP also has an increased flow completion time due to the TCP incast problem [31].

Impact of Flow Size: We fix the number of flows at 3, and Figure 2.4b shows the flow completion time as the flow size increases. We demonstrate that PDQ can better approximate optimal flow completion time as flow size increases. The reason is intuitive: the adverse impact of PDQ inefficiency (e.g., flow initialization latency) on flow completion time becomes relatively insignificant as flow size increases.

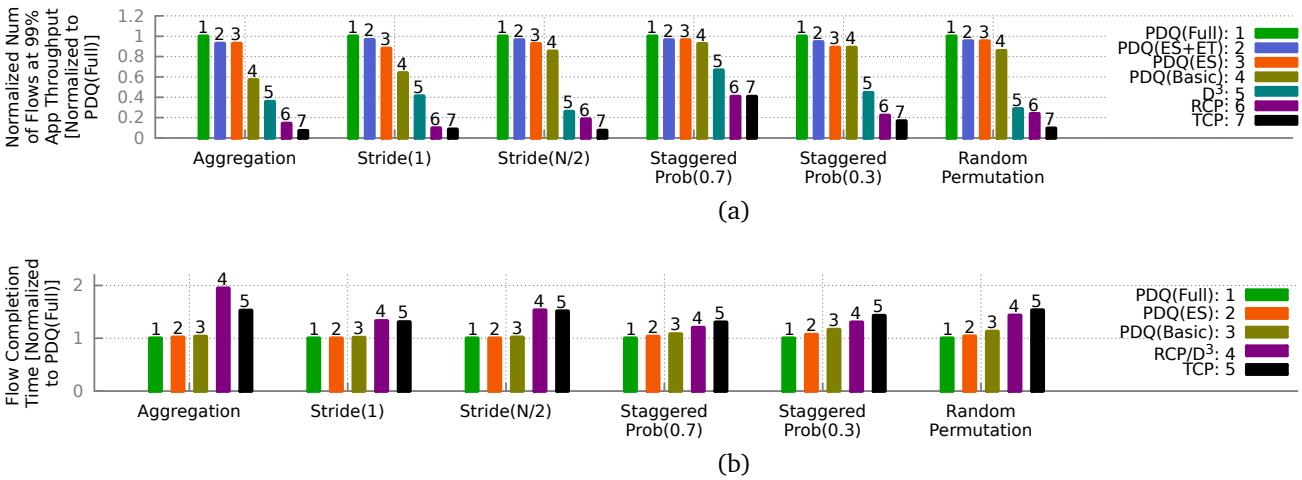


Figure 2.5: PDQ outperforms D^3, RCP and TCP across traffic patterns. (a) Deadline-constrained flows; (b) Deadline-unconstrained flows.

2.5.3 Impact of Traffic Workload

Impact of Sending Pattern: We study the impact of the following sending patterns: **(i)** *Aggregation*: multiple servers send to the same aggregator, as done in the prior experiment. **(ii)** *Stride*(i): a server with index x sends to the host with index $(x + i)$ mod N, where N is the total number of servers; **(iii)** *Staggered Prob*(p): a server sends to another server under the same top-of-rack switch with probability p, and to any other server with probability $1 - p$; **(iv)** *Random Permutation*: a server sends to another randomly-selected server, with a constraint that each server receives traffic from exactly one server (i.e., 1-to-1 mapping).

Figure 2.5 shows that PDQ reaps its benefits across all the sending patterns under consideration. The worst pattern for PDQ is the Staggered Prob(0.7) due to the fact that the variance of the flow RTTs is considerably larger. In this sending pattern, the non-local flows that pass through the core network could have RTTs $3 - 5$ times larger than the local flow RTTs. Thus, the PDQ rate controller, whose update frequency is based on a measurement of *average* flow RTTs, could slightly overreact (or underreact) to flows with relatively large (or small) RTTs. However, even in such a case, PDQ still outperforms the other schemes considerably.

Impact of Traffic Type: We consider two workloads collected from real data centers. First, we use a workload with flow sizes following the distribution from a large-scale commercial data center measured by Greenberg et al. [1]. It

represents a mixture of long and short flows: Most flows are small, and most of the delivered bits are contributed by long flows. In the experiment, we assume that the short flows (with a size of <40 KByte) are deadline-constrained. We conduct these experiments with random permutation traffic.

Figure 2.6a demonstrates that, under this particular workload, PDQ outperforms the other protocols by supporting a significantly higher flow arrival rate. We observed that, in this scenario, PDQ(Full) considerably outperforms PDQ(ES+ET). This suggests that Suppressed Probing plays an important role in minimizing the probing overhead especially when there exists a large collection of paused flows. Figure 2.6b shows that PDQ has lower flow completion time for long flows: a 26% reduction compared with RCP, and a 39% reduction compared with TCP.

We also evaluate performance using a workload collected from a university data center with 500 servers [2]. In particular, we first convert the packet trace, which lasts 10 minutes, to flow-level summaries using Bro [33], then we fed it to the simulator. Likewise, PDQ outperforms other schemes in this regime (Figure 2.6c).

2.5.4 Dynamics of PDQ

Next, we show PDQ's performance over time through two scenarios, each with varying traffic dynamics:

Scenario #1 (Convergence Dynamics): Figure 2.7 shows that PDQ provides seamless flow switching. We assume five flows that start at time 0. The flows have no deadlines, and each flow has a size of ∼1 MByte. The flow size is perturbed slightly such that a flow with smaller index is more critical. Ideally, the five flows together take 40 ms to finish because each flow requires a raw processing time of $\frac{1 \text{ MByte}}{1 \text{ Gbps}} = 8$ ms. With seamless flow switching, PDQ completes at ∼42 ms due to protocol overhead (∼3% bandwidth loss due to TCP/IP header) and first-flow initialization time (two-RTT latency loss; one RTT latency for the sender to receive the SYN-ACK, and an additional RTT for the sender to receive the first DATA-ACK). We observe that PDQ can converge to equilibrium quickly at flow switching time, resulting in a near perfect (100%) bottleneck link utilization (Figure 2.7b). Although an alternative (naive) approach to achieve such high link utilization is to let every flow send with fastest

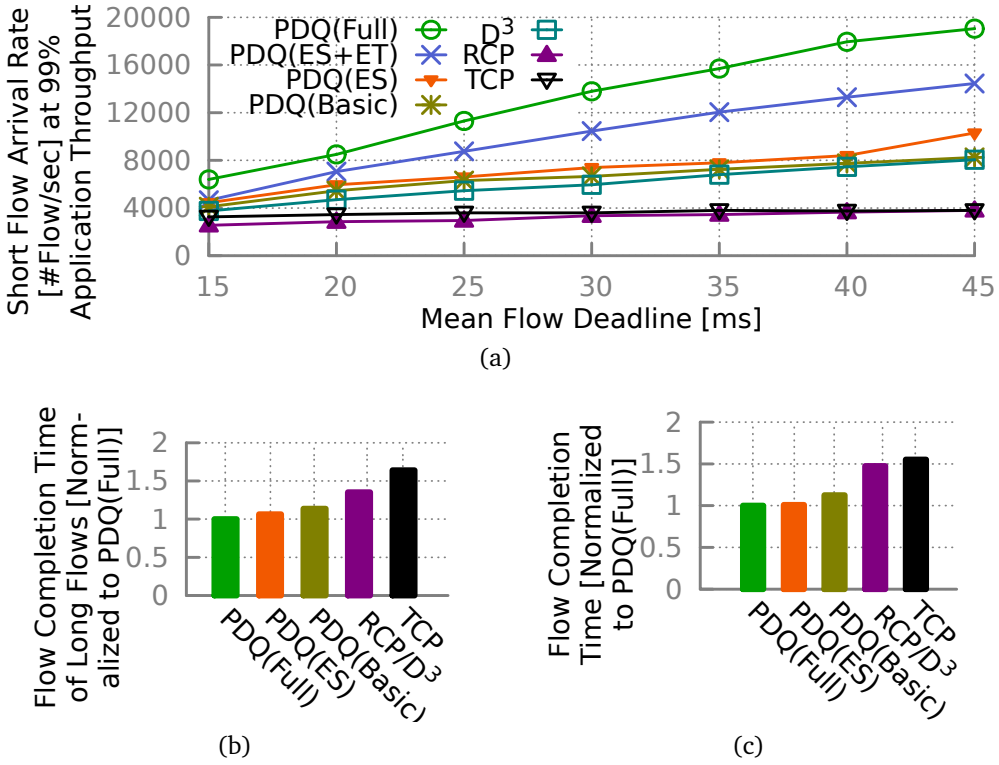
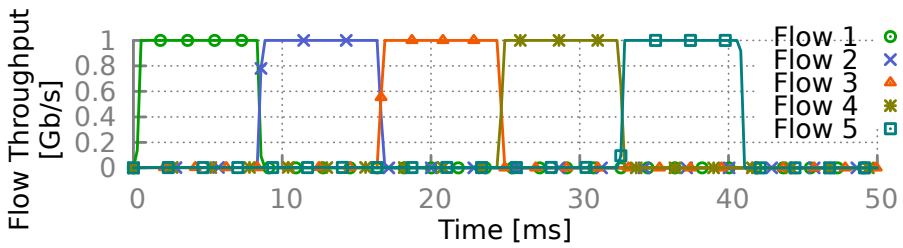


Figure 2.6: Performance evaluation under realistic data center workloads, collected from *(a, b)* a production data center of a large commercial cloud service [1] and *(c)* a university data center located in Midwestern United States (EDU1 in [2]).

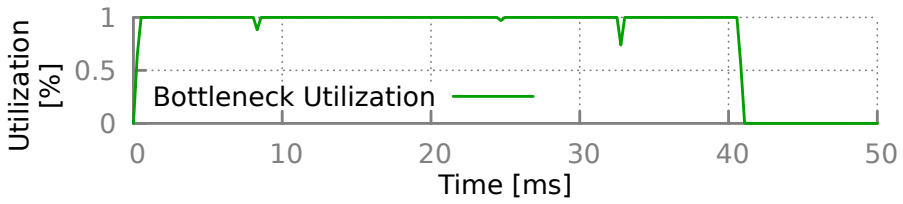
rate, this causes the rapid growth of the queue and potentially leads to congestive packet loss. Unlike this approach, PDQ exhibits a very small queue size[4] and has no packet drops (Figure 2.7c).

Scenario #2 (Robustness to Bursty Traffic): Figure 2.8 shows that PDQ provides high robustness to bursty workloads. We assume a long-lived flow that starts at time 0, and 50 short flows that all start at 10 ms. The short flow sizes are set to 20 KByte with small random perturbation. Figure 2.8a shows that PDQ adapts quickly to sudden bursts of flow arrivals. Because the required delivery time of each short flow is very small ($\frac{20 \text{ KByte}}{1 \text{ Gbps}} \approx 153\ \mu s$), the system never reaches stable state during the preemption period (between 10 and 19 ms). Figure 2.8b shows PDQ adapts quickly to the burst of flows while maintaining high utilization: the average link utilization during the preemption period is 91.7%. Figure 2.8c suggests that PDQ does not compromise the

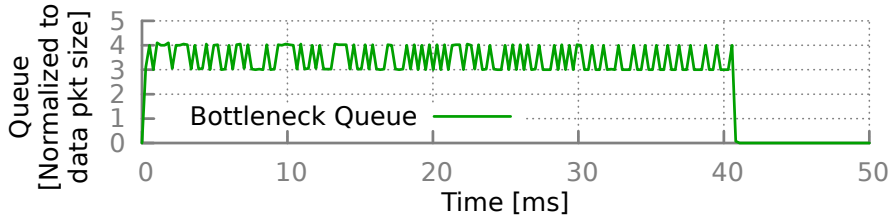
[4]The non-integer values on the y axis comes from the small probing packets.



(a)



(b)



(c)

Figure 2.7: PDQ provides seamless flow switching. It achieves high link utilization at flow switching time, maintains small queue, and converges to the equilibrium quickly.

queue length by having only 5 to 10 packets in the queue, which is about an order of magnitude smaller than what today's data center switches can store. By contrast, XCP in a similar environment results in a queue of ∼60 packets (Figure 11(b) in [19]).

2.5.5 Impact of Network Scale

Today's data centers have many thousands of servers, and it remains unclear whether PDQ will retain its successes at large scales. Unfortunately, our packet-level simulator, which is optimized for high processing speeds, does not scale to large-scale data center topology within reasonable processing time. To study these protocols at large scales, we construct a *flow-level simulator* for PDQ, D^3 and RCP. In particular, we use an iterative approach to find the

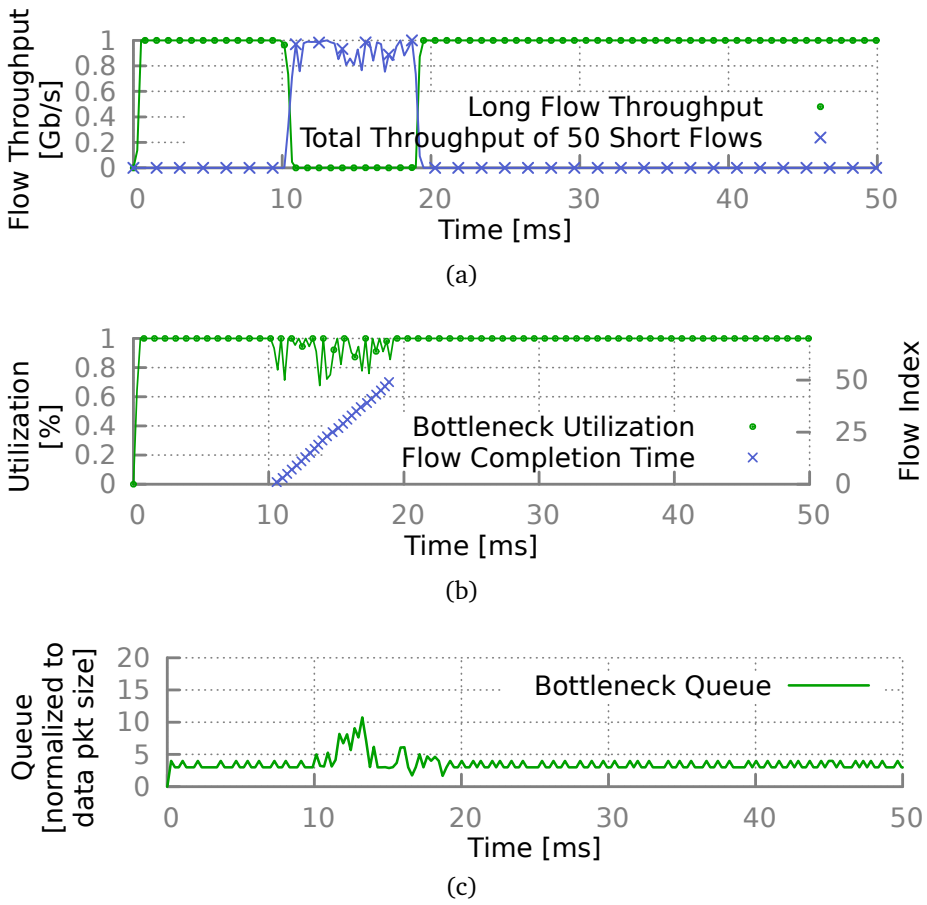


Figure 2.8: PDQ exhibits high robustness to bursty workload. We use a workload of 50 concurrent short flows all start at time 1 ms, and preempting a long-lived flow.

equilibrium flow sending rates with a time scale of 1 ms. The flow-level simulator also considers protocol inefficiencies like flow initialization time and packet header overhead. Although the flow-level simulator does not deal with packet-level dynamics such as timeouts or packet loss, Figure 2.9 shows that, by comparing with the results from packet-level simulation, the flow-level simulation does not compromise the accuracy significantly.

We evaluate three scalable data center topologies: (1) Fat-tree [34], a structured 2-stage Clos network; (2) BCube [35], a server-centric modular network; (3) Jellyfish [36], an unstructured high-bandwidth network using random regular graphs. Figure 2.9 demonstrates that PDQ scales well to large scale, regardless of the topologies we tested. Figure 2.9e shows that about 40% of flow completion times under PDQ are reduced by at least 50% compared to RCP. Only $5-15\%$ of the flows have a larger completion time, and

no more than 0.9% of the flows have 2× completion time.

2.5.6 PDQ Resilience

Resilience to Packet Loss: Next, to evaluate PDQ's performance in the presence of packet loss, we randomly drop packets at the bottleneck link, in both the forward (data) and reverse (acknowledgment) direction. Figure 2.10 demonstrates that PDQ is even more resilient than TCP to packet loss. When packet loss happens, the PDQ rate controller detects anomalous high/low link load quickly and compensates for it with explicit rate control. Thus, packet loss does not significantly affect its performance. For a heavily lossy channel where the packet loss rate is 3% in both directions (i.e., a round-trip packet loss rate of $1-(1-0.03)^2 \approx 5.9\%$), as shown in Figure 2.10b, the flow completion time of PDQ has increased by 11.4%, while that of TCP has significantly increased by 44.7%.

Resilience to Inaccurate Flow Information: For many data center applications (e.g., web search, key-value stores, data processing), previous studies have shown that flow size can be precisely known at flow initiation time.[5] Even for applications without such knowledge, PDQ is resilient to inaccurate flow size information. To demonstrate this, we consider the following two flow-size-unaware schemes. *Random*: the sender randomly chooses a flow criticality at flow initialization time and uses it consistently. *Flow Size Estimation*: the sender estimates the flow size based on the amount of data sent already, and a flow is more critical than another one if it has smaller estimated size. To avoid excessive switching among flows, the sender does not change the flow criticality for every packet it sends. Instead, the sender updates the flow criticality only for every 50 KByte it sends. Figure 2.11 demonstrates two important results: (i) PDQ does require reasonable estimate of flow size as random criticality can lead to large mean flow completion time in heavy-tailed flow size distribution. (ii) With a simple estimation scheme, PDQ still compares favorably against RCP in both uniform and heavy-tailed flow size distributions.

[5] See the discussion in §2.1 of [13].

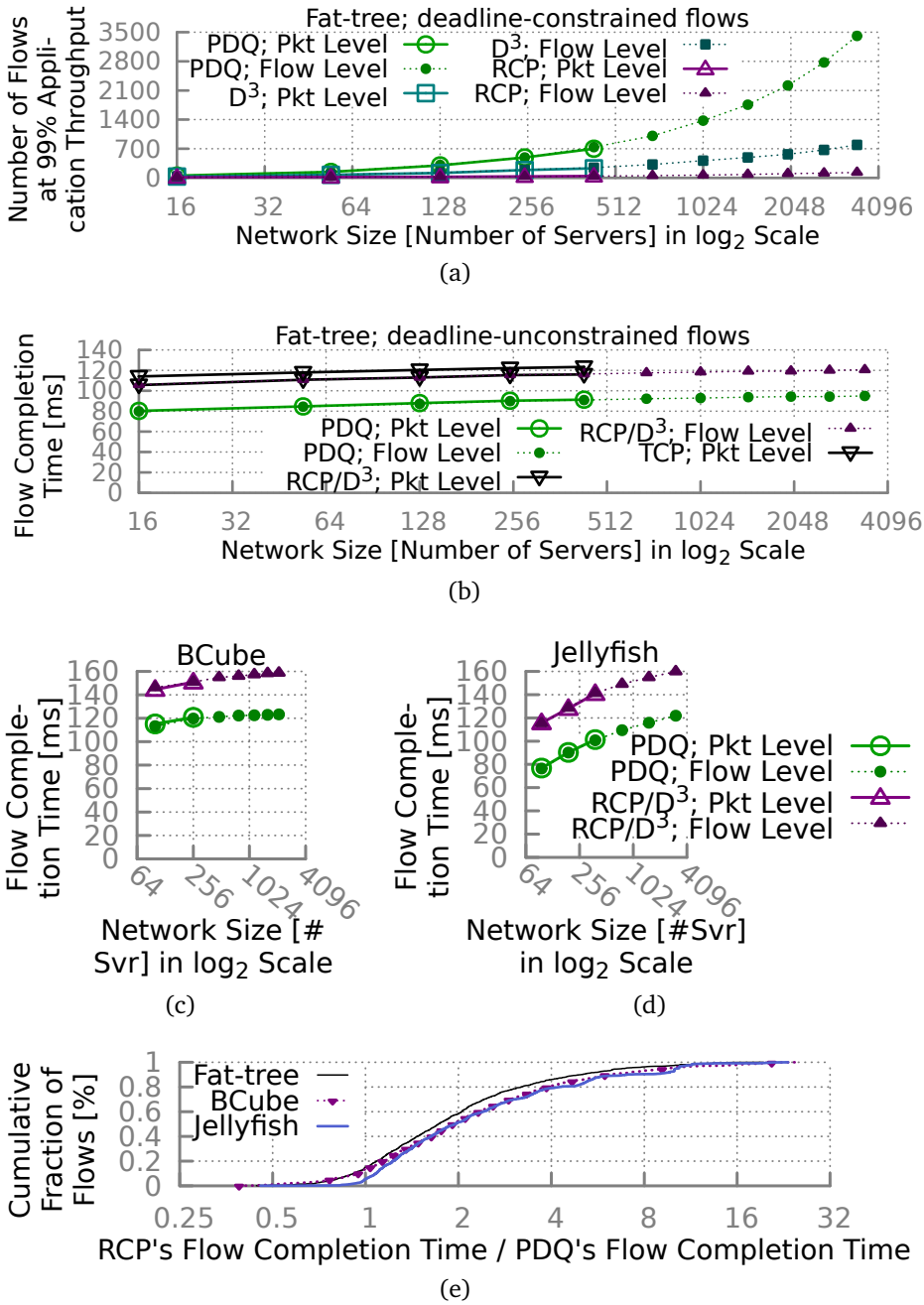


Figure 2.9: PDQ performs well across a variety of data center topologies. (a,b) Fat-tree; (c) BCube with dual-port servers; (d) Jellyfish with 24-port switches, using a 2:1 ratio of network port count to server port count. (e) For network flows, the ratio of the flow completion time under PDQ to the flow completion time under RCP (flow-level simulation; # servers is ∼128). All experiments are carried out using random permutation traffic; top figure: deadline-constrained flows; bottom four figures: deadline-unconstrained flows with 10 sending flows per server.

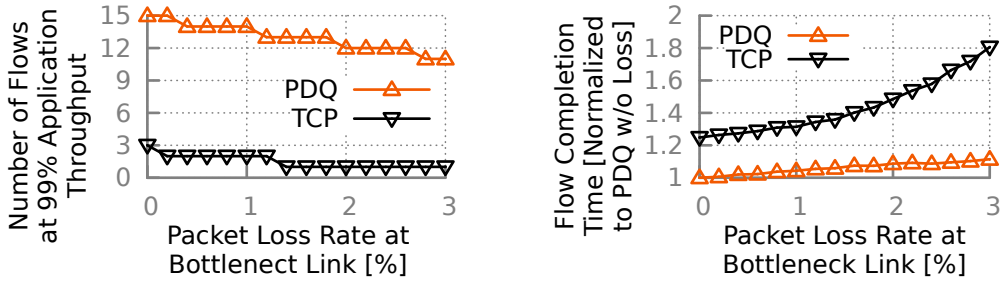


Figure 2.10: PDQ is resilient to packet loss in both forward and reverse directions: (a) deadline-constrained and (b) deadline-unconstrained cases. Query aggregation workload.

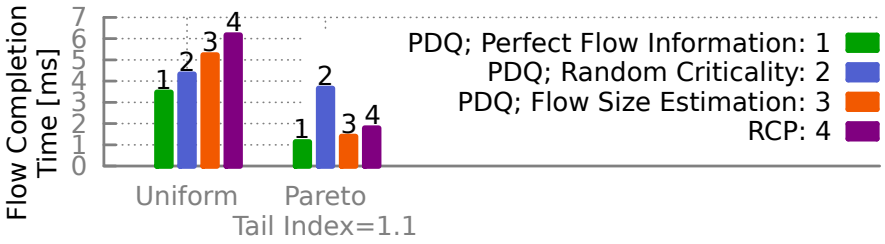


Figure 2.11: PDQ is resilient to inaccurate flow information. For PDQ without flow size information, the flow criticality is updated for every 50 KByte it sends. Query aggregation workload, 10 deadline-unconstrained flows with a mean size of 100 KByte. Flow-level simulation.

2.6 Multipath forwarding

Several recent works [37,38] show the benefits of multipath TCP, ranging from improved reliability to higher network utilization. Motivated by this work, we propose Multipath PDQ (M-PDQ), which enables a single flow to be striped across multiple network paths.

When a flow arrives, the M-PDQ sender splits the flow into multiple subflows, and sends a SYN packet for each subflow. To minimize the flow completion time, the M-PDQ sender periodically shifts the load from the paused subflows to the sending one with the minimal remaining load. To support M-PDQ, the switch uses flow-level Equal-Cost MultiPath (ECMP) to assign subflows to paths. The PDQ switch requires no additional modification except ECMP. The M-PDQ receiver maintains a single shared buffer for a multipath flow to resequence out-of-order packet arrivals, as done in Multipath TCP [37].

We illustrate the performance gains of M-PDQ using BCube [35], a data cen-

ter topology that allows M-PDQ to exploit the path diversity between hosts. We implement BCube address-based routing to derive multiple parallel paths. Using random permutation traffic, Figure 2.12a demonstrates the impact of the system load on flow completion time of M-PDQ. Here, we split a flow into 3 M-PDQ subflows. Under light loads, M-PDQ can reduce flow completion time by a factor of two. This happens because M-PDQ exploits more links that are underutilized or idle than single-path PDQ. As load increases, these advantages are reduced, since even single-path PDQ can saturate the bandwidth of nearly all links.

However, as shown in Figure 2.12a, M-PDQ still retains its benefits because M-PDQ allows a critical flow to have higher sending rate by utilizing multiple parallel paths. Finally, we fix the workload at 100% to stress the network (Figures 2.12b and 2.12c). We observe that M-PDQ needs about 4 subflows to reach 97% of its full potential. By allowing servers to use all four interfaces (whereas single-path PDQ can use only one), M-PDQ provides a significant performance improvement.

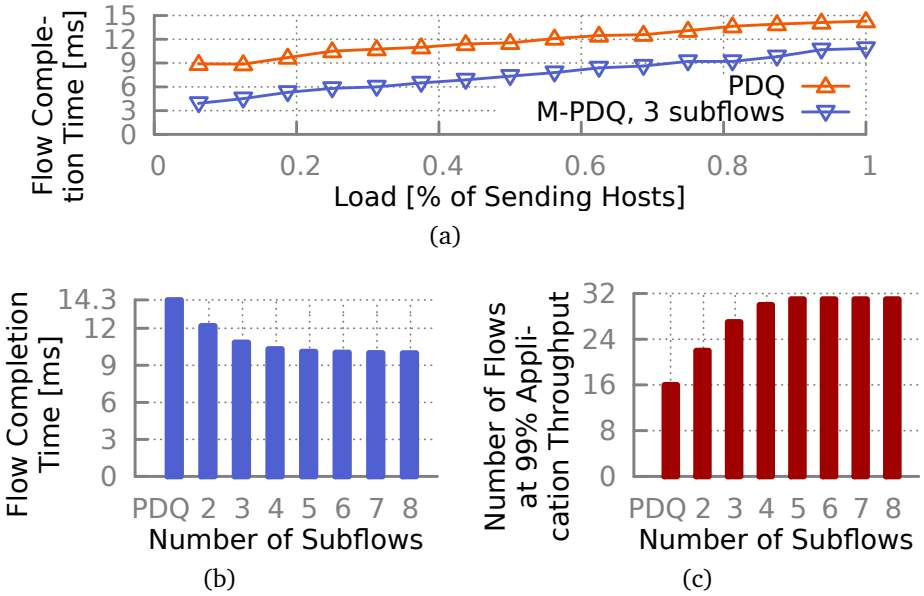


Figure 2.12: Multipath PDQ achieves better performance. BCube(2,3) with random permutation traffic. (a, b) deadline-unconstrained, (c) deadline-constrained flows.

2.7 Discussion

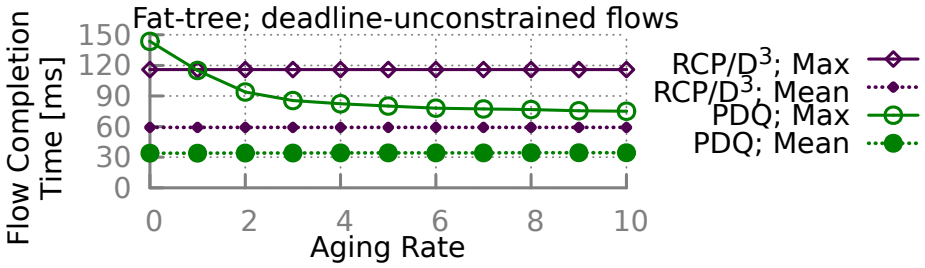
Flow Size Estimation. For many data center applications (e.g., web search, key-value stores, data processing), previous studies have shown that flow size can be precisely known at flow initiation time.[6] Even for applications without such knowledge, there are good schemes to accurately estimate flow sizes, for example by matching based on packet header fields, by applying machine learning techniques to classify flows [39], or by monitoring the server-side buffer [40]. The sender can also estimate the flow size based on the amount of data sent already, as done in [28]. We also demonstrate that PDQ preserves nearly all its performance gains even given inaccurate flow information (§2.5.6).

Fairness. One could argue the performance gains of PDQ over other protocols stem from the fact that PDQ unfairly penalizes less critical flows. Perhaps counter-intuitively, the performance gain of SJF over fair sharing does not usually come at the expense of long jobs. An analysis [26] shows that at least 99% of jobs have a smaller completion time under SJF than under fair sharing, and this percentage increases further when the traffic load is less than half.[7] Our results further demonstrate that, even in complex data center networks with thousands of concurrent flows and multiple bottlenecks, $85-95\%$ of PDQ's flows have a smaller completion time than RCP, and the worst PDQ flow suffers an inflation factor of only 2.57 as compared with RCP (Figure 2.9e). Moreover, unfairness might not be a primary concern in data center networks where the network is owned by a single entity that has full control of flow criticality. However, if desired, the operator can easily override the flow comparator to achieve a wide range of goals, including fairness. For example, to prevent starvation, the operator could gradually increase the criticality of a flow based on its waiting time. Using a fat-tree topology with 256 servers, Figure 2.13 demonstrates that this "flow aging" scheme is effective, reducing the worst flow completion time by $\sim$48%, while the mean flow completion time increases only 1.7%.

When flow completion time is not the priority. Flow completion time is not the best metric for some protocols. For example, real-time audio and video may require the ability to *stream*, or provide a number of flows with a

[6] See the discussion in §2.1 of [13].

[7] Assuming a M/G/1 queueing model with heavy-tailed flow distributions; see [26].



(a)

Figure 2.13: Aging helps prevent less critical flows from starvation and shortens their completion time. The PDQ sender increases flow criticality by reducing $\mathscr{T}_H$ by a factor of $2^{\alpha t}$, where α is a parameter that controls the aging rate, and t is the flow waiting time (in terms of 100 ms). Flow-level simulation; 128-server fat-tree topology; random permutation traffic.

fixed fraction of network capacity. For these applications, protocols designed for streaming transport may be a better fit. One can configure the rate controller (§2.3.3) to slice the network into PDQ-traffic and non-PDQ-traffic, and use some other transport protocol for non-PDQ-traffic. In addition, there are also applications where the receiver may not be able to process incoming data at the full line rate. In such cases, sending any rate faster than what receiver can process does not offer substantial benefits. Assuming the receiver buffers are reasonably small, PDQ will back off and allocate remaining bandwidth to another flow.

Does preemption in PDQ require rewriting applications? A preempted flow is paused (briefly), not terminated. From the application's perspective, it is equivalent to TCP being slow momentarily; the transport connection stays open. Applications do not need to be rewritten since preemption is hidden in the transport layer.

Incentive to game the system. Users are rational and may have an incentive to improve the completion time of their own flows by splitting each flow into small flows. While a similar issue happens to D^3, TCP and RCP[8], users in PDQ may have an even greater incentive, since PDQ does preemption. It seems plausible to penalize users for having a large number of short flows by reducing their flows' criticality. Developing a specific scheme remains as future work.

[8]In TCP/RCP, users may achieve higher aggregated throughput by splitting a flow into smaller flows; in D^3, users may request a higher rate than the flow actually needs.

Deployment. On end hosts, one can implement PDQ by inserting a shim layer between the IP and the transport layers. In particular, the sender maintains a set of PDQ variables, intercepts all calls between IP and transport layer, attaches and strips off the PDQ scheduling header[9], and passes the packet segment to IP/transport layer accordingly. Additionally, the shim layer could provide an API that allows applications to specify the deadline and flow size, or it could avoid the API by estimating flow sizes (§2.5.6). The PDQ sender can easily override TCP's congestion window size to control the flow sending rate. We note that PDQ requires only a few more variables per flow on end hosts. On switches, similar to previous proposals such as D^3, a vendor can implement PDQ by making modifications to the switch's hardware and software. Per-packet operations like modifying header fields are already implemented on most vendors' hardware (e.g., ASICs), which can be directly used by our design. The more complex operations like computing the aggregated flow rate and sorting/updating the flow list can be implemented in software. We note that PDQ's per-packet running time is $O(\kappa)$ for the top κ flows and $O(1)$ for the rest of the flows, where κ is a small number of flows with the highest criticality and can be bounded as in §2.3.3. The majority of the sending flows' scheduling headers would remain unmodified[10] by switches.

2.8 Related work

D^3: While D^3 [13] is a deadline-aware protocol that also employs explicit rate control like PDQ, it neither resequences flow transmission order nor preempts flows, resulting in a substantially different flow schedule which serves flows according to the order of their arrival. Unfortunately, this allows flows with large deadlines to hog the bottleneck bandwidth, blocking short flows that arrived later.

Fair Sharing: TCP, RCP [20] and DCTCP [24] all emulate fair sharing, which leads to suboptimal flow completion time.

[9]The 16-byte scheduling header consists of 4 fields, each occupying 4 bytes: $\mathscr{R}_H$, $\mathscr{P}_H$, $\mathscr{D}_H$, and $\mathscr{T}_H$. The PDQ receiver adds $\mathscr{I}_S$ and RTT_S to the header by reusing the fields used by $\mathscr{D}_H$ and $\mathscr{T}_H$. This is feasible because $\mathscr{D}_H$ and $\mathscr{T}_H$ are used only in the forward path, while $\mathscr{I}_S$ and RTT_S are used only in the reverse path. Any reasonable hashing that maps switch ID to 4-byte $\mathscr{P}_H$ should provide negligible collision probability.

[10]Until, of course, the flow is preempted or terminated.

TCP/RCP with Priority Queueing: One could use priority queuing at switches and assigning different priority levels to flows based on their deadlines. Previous studies [13] showed that, using two-level priorities, TCP/RCP with priority queueing suffers from losses and falls behind D^3, and increasing the priority classes to four does not significantly improve performance. This is because flows can have very different deadlines and require a large number of priority classes, while switches nowadays provide only a small number of classes, mostly no more than ten.

ATM: One could use ATM to achieve QoS priority control. However, ATM's CLP classifies traffic into only two priority levels), while PDQ gives each flow a unique priority. Moreover, ATM is unable to preempt flows (i.e., new flows cannot affect existing ones).

DeTail: Zats et al. propose DeTail [12], an in-network multipath-aware congestion management mechanism that reduces the flow completion time "tail" in datacenter networks. However, it targets neither mean flow completion time nor the number of deadline-missing flows. Unlike DeTail which removes the tail, PDQ can save ∼30% flow completion time on average (compared with TCP and RCP), reducing the completion time of almost every flow (e.g., $85\% - 95\%$ of the flows, Figure 2.9e).

2.9 Conclusion

We proposed PDQ, a flow scheduling protocol designed to complete flows quickly and meet flow deadlines. PDQ provides a distributed algorithm to approximate a range of scheduling disciplines based on relative priority of flows, minimizing mean flow completion time and the number of deadline-missing flows. We perform extensive packet-level and flow-level simulation of PDQ and several related works, leveraging real datacenter workloads and a variety of traffic patterns, network topologies, and network sizes. We find that PDQ provides significant advantages over existing schemes. In particular, PDQ can reduce by ∼30% the average flow completion time as compared with TCP, RCP and D^3; and can support $3\times$ as many concurrent senders as D^3 while meeting flow deadlines. We also design a multipath variant of PDQ by splitting a single flow into multiple subflows, and demonstrate that M-PDQ achieves further performance and reliability gains under a variety of settings.

CHAPTER 3

ACHIEVING HIGH UTILIZATION IN INTER-DATACENTER WANS

In this chapter, we present a *Software Defined Transport* architecture (SDT), where a central transport controller schedules and dynamically re-schedules the flow sending rates and network forwarding place. We then build a software-driven flow controller in the context of inter-datacenter WANs. Unlike previous chapter, which assumes the switches can be modified to compute a flow schedule collaboratively, this chapter shows how to achieve similar flexibility using a software-based controller that requires no hardware changes to switches.

We first present the service requirements in inter-datacenter WANs. Then we give an outline of our design to demonstrate how to apply SDT to inter-datacenter WANs to meet these service requirements. A key design is a scalable algorithm for global allocation that optimizes network utilization subject to constraints on service priority and fairness. To scale up to global inter-datacenter networks, we take a practical approach to approximately compute the rate allocation with provably performance deviation bound to improve the computational time. We develop and evaluate our approach via prototype and simulation using real production inter-datacenter traces. We found SDT, by centrally controlling when and how much traffic each network service sends and frequently re-optimizing network data plane configuration, carries significantly more traffic than today's traffic engineering practice in inter-datacenter WAN while satisfying the service requirements.

3.1 Background

The wide area network (WAN) that connects the datacenters (DC) is critical infrastructure for cloud and online services providers such as Amazon, Google, and Microsoft. Inter-DC transfers are the lifeblood of many services; for per-

formance and reliability, updates that are generated by customers or internal processes at one of the DCs need timely mirroring to others. Given the need for high capacity—traffic between DCs is a significant fraction of Internet traffic and rapidly growing [41]—and unique traffic characteristics, the inter-DC WAN is often a dedicated network, distinct from the WAN that connects to ISPs to help reach end users [42]. It is an expensive resource, with amortized annual cost of 100s of millions of dollars, as it provides 100s of Gbps to Tbps of capacity over long distances.

However, providers are unable to fully leverage this investment. Inter-DC WANs have extremely poor efficiency; the average utilization of even the busier links is 40-60%. One culprit is the lack of coordination among the services that use the network. Barring coarse, static limits in some cases, services send traffic whenever they want and however much they want. As a result, the network cycles through periods of peaks and troughs. Since it must be provisioned for peak usage to avoid congestion, the network is under-subscribed on average.

We observe that network usage does not have to be this way if we can exploit service characteristics. For instance, some inter-DC services are tolerant of delays. The cyclical behavior can be tamped if their traffic is sent when the demand from other services is low. This coordination will boost average utilization and enable the network to either carry more traffic with the same capacity or use less capacity to carry the same traffic.

Another culprit behind poor efficiency is the distributed resource allocation model of today, typically implemented using MPLS-TE (Multi Protocol Label Switching Traffic Engineering) [43, 44]. In this model, no entity has a global view and routers greedily select paths for the traffic they source. As a result, the network can get stuck in locally optimal routing patterns that are globally suboptimal [45].

We present SWAN (Software-driven WAN), a resource controller that enables inter-DC WANs to carry significantly more traffic. By itself, carrying more traffic is easy—we can let loose bandwidth-hungry services. SWAN achieves high efficiency while meeting policy goals such as preferential treatment for higher-priority services and fairness among equal-priority services. Per observations above, its two key aspects are i) coordinating the sending rates of services; and ii) centrally allocating network resources. Based on the current demands of services, the SWAN controller decides how much each service can send,

and using SDN (Software-defined networking) capabilities, it configures the network's data plane to carry that traffic.

While many recent works have developed centralized resource allocators, most of them target *intra*-DC local area networks [15, 28, 46–50], where the challenges and opportunities are different. Intra-DC networks have many more links and switches, but they also have structured topologies and low RTTs. One exception is Google's recent announcement on using SDN to manage their inter-DC WAN [42]. But the challenges they faced and details of their design are not known (publicly).

We develop a prototype of SWAN, and evaluate our approach through testbed experiments and simulations using traffic and topology data from two production inter-DC WANs. We find that SWAN carries 98% of traffic carried by an optimal method that is not hindered by rule capacity limits and has no overhead related to network updates. This traffic is 60% more than what MPLS-TE carries. We also show that changes to network updates are quick, requiring only 1-3 steps.

While this chapter focuses on inter-DC WANs, many of our underlying techniques are applicable to other WANs such as ISP networks. We show that even without controlling how much traffic services send, an ability that is unique to the inter-DC context, our techniques for global resource and change management allow the network to carry 16-25% more traffic than MPLS-TE.

3.2 Motivation

Inter-DC WANs carry traffic from a range of services, where a *service* is an activity across multiple hosts. Externally visible functionality is usually enabled by multiple internal services (e.g., search may use Web-crawler, index-builder, and query-responder services). Prior work [51] and our conversations with operators reveal that services fall into three broad types, based on their performance requirements.

Interactive services are in the critical path of end user experience. An example is when one DC contacts another in the process of responding to a user request because not all information is available in the first DC. Interactive traffic is highly sensitive to loss and delay; even small increases in response time (100 ms) degrade user experience [13].

Elastic services are not in the critical path of user experience but still require timely delivery. An example is replicating a data update to another DC. Elastic traffic requires delivery within a few seconds or minutes. The consequences of delay vary with the service. In the replication example, the risk is loss of data if a failure occurs or that a user will observe data inconsistency.

Background services conduct maintenance and provisioning activities. An example is copying all the data of a service to another DC for long-term storage or as a precursor to running the service there. Such traffic tends to be bandwidth hungry. While it has no explicit deadline or a long deadline, it is still desirable to complete transfers as soon as possible—delays lower business agility and tie up expensive server resources.

3.2.1 Current traffic engineering practice

Many WANs are operated using MPLS TE today. To effectively use network capacity, MPLS TE spreads traffic across a number of tunnels between ingress-egress router pairs. Ingress routers split traffic, typically equally using equal cost multipath routing (ECMP), across the tunnels to the same egress. They also estimate the traffic demand for each tunnel and find network paths for it using the constrained shortest path first (CSPF) algorithm, which identifies the shortest path that can accommodate the tunnel's traffic (subject to priorities; see below).

With MPLS TE, service differentiation can be provided using two mechanisms. First, tunnels are assigned priorities and different types of services are mapped to different tunnels. Higher priority tunnels can displace lower priority tunnels and thus obtain shorter paths; the ingress routers of displaced tunnels must then find new paths. Second, packets carry differentiated services code point (DSCP) bits in the IP header. Switches map different bits to different priority queues, which ensures that packets are not delayed or dropped due to lower-priority traffic; they may still be delayed or dropped due to equal or higher priority traffic. Switches typically have only a few priority queues (4–8).

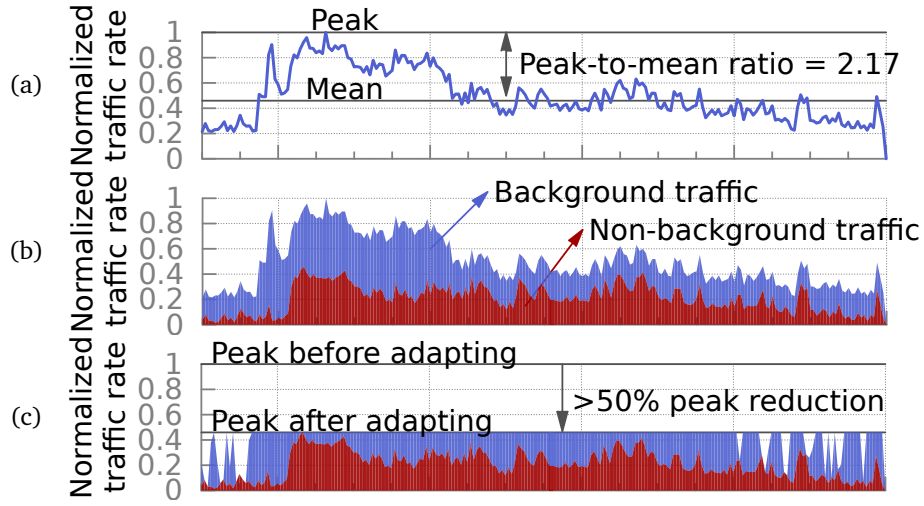


Figure 3.1: Illustration of poor utilization. (a) Daily traffic pattern on a busy link in a production inter-DC WAN. (b) Breakdown based on traffic type. (c) Reduction in peak usage if background traffic is dynamically adapted.

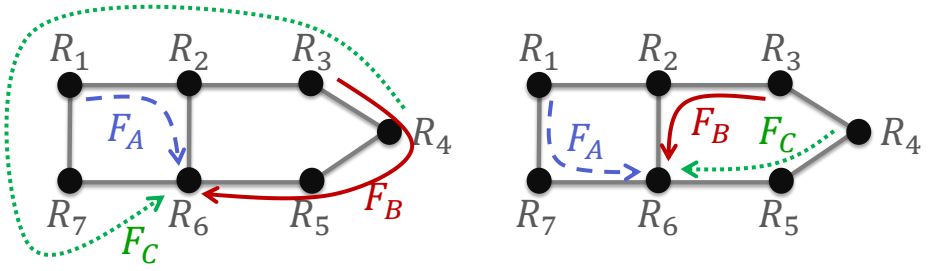
3.2.2 Problems of MPLS TE

Inter-DC WANs suffer from two key problems today.

Poor efficiency: The amount of traffic the WAN carries tends to be low compared to capacity. For a production inter-DC WAN, which we call IDN (§3.5.1), we find that the average utilization of half the links is under 30% and of three in four links is under 50%.

Two factors lead to poor efficiency. First, services send whenever and however much traffic they want, without regard to the current state of the network or other services. This lack of coordination leads to network swinging between over- and under-subscription. Figure 3.1a shows the load over a day on a busy link in IDN. Assuming capacity matches peak usage (a common provisioning model to avoid congestion), the average utilization on this link is under 50%. Thus, half the provisioned capacity is wasted. This inefficiency is not fundamental but can be remedied by exploiting traffic characteristics. As a simple illustration, Figure 3.1b separates background traffic. Figure 3.1c shows that the same total traffic can fit in half the capacity if background traffic is adapted to use capacity left unused by other traffic.

Second, the local, greedy resource allocation model of MPLS TE is inefficient. Consider Figure 3.2 in which each link can carry at most one flow. If the flows arrive in the order F_A, F_B, and F_C, Figure 3.2a shows the path assignment with MPLS TE: F_A is assigned to the top path which is one of the



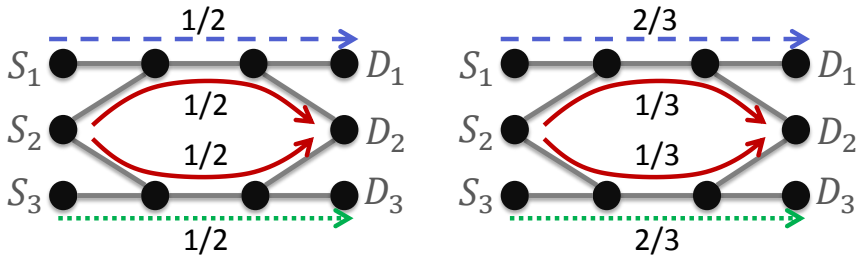
(a) Local path selection (b) Globally optimal paths

Figure 3.2: Inefficient routing due to local allocation.

shortest paths; when F_B arrives, it is assigned to the shortest path with available capacity (CSPF); and the same happens with F_C. Figure 3.2b shows a more efficient routing pattern with shorter paths and many links freed up to carry more traffic. Such an allocation requires non-local changes, e.g., moving F_A to the lower path when F_B arrives.

Partial solutions for such inefficiency exist. Flows can be split across two tunnels, which would divide F_A across the top and bottom paths, allowing half of F_B and F_C to use direct paths; a preemption strategy that prefers shorter paths can also help. But such strategies do not address the fundamental problem of local allocation decisions [45].

Poor sharing: Inter-DC WANs have limited support for flexible resource allocation. For instance, it is difficult to be fair across services or favor some services over certain paths. When services compete today, they typically obtain throughput proportional to their sending rate, an undesirable outcome (e.g., it creates perverse incentives for service developers). Mapping each service onto its own queue at routers can alleviate problems but the number of services (100s) far exceeds the number of available router queues. Even if we had infinite queues and could ensure fairness on the data plane, network-wide fairness is not possible without controlling which flows have access to which paths. Consider Figure 3.3 in which each link has unit capacity and each service ($S_i \rightarrow D_i$) has unit demand. With link-level fairness, $S_2 \rightarrow D_2$ gets twice the throughput of other services. As we show, flexible sharing can be implemented with a limited number of queues by carefully allocating paths to traffic and control the sending rate of services.



(a) Link-level (b) Network-wide

Figure 3.3: Link-level fairness ≠ network-wide fairness.

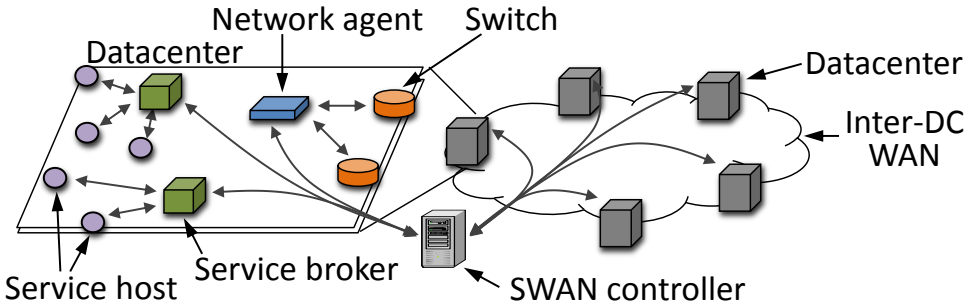


Figure 3.4: Architecture of SDT.

3.3 Design

Figure 3.4 shows the architecture of SDT. A logically centralized *controller* orchestrates the WAN. Each non-interactive service has a *broker* that aggregates demands from the hosts and apportions the allocated rate to them. One or more *network agents* intermediate between the controller and the switches. This architecture provides scale—by providing parallelism where needed—and choice—each service can implement a rate allocation strategy that is most suited for it.

Service hosts and brokers collectively estimate the service's current demand and limit it to the rate allocated by the controller. They can implement this functionality however they see fit. Our current implementation draws on distributed rate limiting [47]. A shim in the host OS estimates its demand to each remote DC for the next T_h (10 seconds) and asks the broker for an allocation. It then uses a token bucket per remote DC to enforce the allocated rate and tags packets with DSCP bits to indicate the service's priority class.

The service broker aggregates demand from hosts and updates the controller every T_s (5 minutes). It apportions its allocation from the controller piecemeal,

in time units of T_h, to hosts in a proportionally fair manner. This way, T_h is the maximum time that a newly arriving host has to wait before starting to transmit. It is also the maximum time a service takes to change its sending rate to a new allocation. Brokers that suddenly experience radically larger demands can ask for more any time; the controller does a lightweight computation to determine how much of the additional demand can be carried without altering network configuration.

Network agents track topology and traffic with the aid of switches. They relay news about topology changes to the controller right away and collect and report information about traffic, at the granularity of OpenFlow rules, every T_a=5 minutes. They are also responsible for reliably changing switch rules as requested by the controller. Before returning success, an agent reads the relevant part of the switch rule table to ensure that the changes have been successfully applied.

SWAN controller uses the current information on service demands and network topology to do the following every T_c=5 minutes.

1. Compute the service allocations and forwarding plane configuration for the network (§3.3.1, §3.3.2).

2. Signal new allocations to services whose allocation has decreased. Wait for T_h seconds for the service to lower its sending rate.

3. Change the forwarding state (details in §4.2) and then signal the new allocation to the remaining services (whose allocation has increased).

3.3.1 Forwarding plane configuration

SWAN uses label-based forwarding. Doing so reduces forwarding complexity; the complex classification that may be required to assign a label to traffic is done just once, at the source switch. Remaining switches simply read the label and forward the packet based on the rules for that label. We currently use VLAN IDs as labels.

Source switches also split traffic across multiple tunnels (labels). We propose to implement unequal splitting, which leads to more efficient allocation [52], using *group tables* in the OpenFlow pipeline. The first table maps the packet, based on its destination and other characteristics (e.g., DSCP bits),

to a group table. Each group table consists of the set of tunnels available and a weight assignment that reflects the ratio of traffic to be sent to each tunnel. Conversations with switch vendors indicate that most will roll out support for unequal splitting. When such support is unavailable, SWAN uses traffic profiles to pick boundaries in the range of IP addresses belonging to a DC such that splitting traffic to that DC at these boundaries will lead to the desired split. Then, SWAN configures rules at the source switch to map IP destination spaces to tunnels. Our experiments with traffic from a production WAN show that implementing unequal splits in this way leads to a small amount of error (less than 2%).

3.3.2 Computing service allocations

When computing allocated rate for services, our goal is to maximize network utilization subject to service priorities and approximate max-min fairness among same-priority services. The allocation process must be scalable enough to handle WANs with 100s of switches.

Inputs: The allocation uses as input the service demands d_i between pairs of DCs. While brokers report the demand for non-interactive services, SWAN estimates the demand of interactive services (see below). We also use as input the paths (tunnels) available between a DC pair. Running an unconstrained multi-commodity problem could result in allocations that require many rules at switches. Since a DC pair's traffic could flow through any link, every switch may need rules to split every pair's traffic across its outgoing ports. Constraining the usable paths avoids this possibility and also simplifies updates to network configuration (§4.2). But it may lead to lower overall throughput. For our two production inter-DC WANs, we find that using the 15 shortest paths between each pair of DCs results in negligible loss of throughput.

Allocation LP: Figure 4 shows the LP used in SWAN. At the core is the MCF (multi-commodity flow) function that maximizes the overall throughput while preferring shorter paths; ϵ is a small constant and tunnel weights w_j are proportional to latency. s_{Pri} is the fraction of scratch link capacity that enables congestion-managed network updates; it can be different for different priority classes and this will be used in §4.2. The SWAN Allocation function allocates rate by invoking MCF separately for classes in priority order. After a class is

Inputs:

d_i: flow demands for source destination pair i
w_j: weight of tunnel j (e.g., latency)
c_l: capacity of link l
s_{Pri}: scratch capacity ($[0, 50\%]$) for class Pri
$I_{j,l}$: binary; indicates if tunnel j uses link l

Outputs:

$b_i = \sum_j b_{i,j}$: allocation of flow i over tunnel j

Func: Allocation:
$\forall$ links l : $c_l^{remain} \leftarrow c_l$;
for $Pri =$ Interactive, Elastic, ..., Background **do**
  $\{b_i\} \leftarrow \begin{matrix}\text{Throughput Maximization} \\ \text{Approx. Max-Min Fairness}\end{matrix} \left(Pri, \{c_l^{remain}\}\right)$;
  $c_l^{remain} \leftarrow c_l^{remain} - \sum_{i,j} b_{i,j} \cdot I_{j,l}$
end

Func: Throughput Maximization$(Pri, \{c_l\})$:
return MCF$(Pri, \{c_l\}, s, 0, \infty, \emptyset)$;

Func: Approx. Max-Min Fairness$(Pri, \{c_l\})$:
// $\alpha > 1$ and $U > 0$ trade-off unfairness for runtime $T \leftarrow \lceil \log_\alpha \frac{\max(d_i)}{U} \rceil$;
$F \leftarrow \emptyset$;
for $k = 1 \ldots T$ **do**
  foreach $b_i \in$ MCF$(Pri, \{c_l\}, s, \alpha^{k-1}U, \alpha^k U, F)$ **do**
    if $i \notin F$ **and** $b_i < \min(d_i, \alpha^k U)$ **then**
      $F \leftarrow F + \{i\}$; $f_i \leftarrow b_i$; // flow saturated
    end
  end
end
return $\{f_i : i \in F\}$;

Func: MCF$(Pri, \{c_l\}, b_{Low}, b_{High}, F)$:

$$\begin{aligned}
\text{maximize} \quad & \textstyle\sum_i b_i - \epsilon(\sum_{i,j} w_j \cdot b_{i,j}) \\
\text{subject to} \quad & \forall i \notin F : b_{Low} \le b_i \le \min(d_i, b_{High}); \\
& \forall i \in F : b_i = f_i; \\
& \forall l : \textstyle\sum_{i,j} b_{i,j} \cdot I_{j,l} \le (1 - s_{Pri}) c_l; \\
& \forall (i,j) : b_{i,j} \ge 0.
\end{aligned}$$

Algorithm 4: Computing allocations over a set of tunnels.

allocated, its allocation is removed from remaining link capacity.

It is easy to see that our allocation respects traffic priorities. By allocating demands in priority order, SWAN also ensures that higher priority traffic is like-

lier to use shorter paths. This ordering also keeps the computation simple (as MCF's complexity increases manifold with the number of constraints). While, in general, it may reduce overall utilization, in practice, SWAN achieves nearly optimal utilization (§3.5).

Max-min fairness in SWAN can be achieved by wrapping the LP with a loop that always assigns the minimum flows in polynomial time [53]. However, this extra loop turns out to be rather costly in practice, so we decided to go for an approximative solution instead. SWAN provides approximate max-min fairness for services in the same class by invoking MCF in T steps, with the constraint that at step k, flows are allocated rates in the range $\left[\alpha^{k-1}U, \alpha^{k}U\right]$, but no more than their demand. A flow's allocation is *frozen* at step k when it is allocated its full demand d_i at that step or it receives a rate smaller than $\alpha^{k}U$ due to capacity constraints. With this algorithm, we can prove that if r_i and b_i are the max-min fair rate and the rate allocated to flow i, and $U \leq \min_i(r_i)$, then MCF is an α-approximation algorithm, i.e., $b_i \in \left[\frac{r_i}{\alpha}, \alpha r_i\right]$ (Theorem 1 in Appendix).

Many proposals exist to combine network-wide max-min fairness with high throughput. A recent one offers a search function that is shown to empirically reduce the number of LPs that need to be solved [54]. Our contribution is showing that one can trade-off the number of LP calls and the degree of unfairness. The number of LPs we solve per priority is T; with $\max d_i$=10Gbps, U=10Mbps and α=2, we get T=10. We find that SWAN's allocations are highly fair and take less than a second combined for all priorities (§3.5). In contrast, the proposal above reports running times of over a minute.

Finally, our approach can be easily extended to other policy goals such as virtually dedicating capacity to a flow over certain paths and weighted max-min fairness.

Post-processing: The solution produced by the LP may not be feasible to implement; while it obeys link capacity concerns, it disregards rule count limits on switches. Directly including these limits in the LP would turn the LP into an Integer LP making it intractably complex. Hence, SWAN post-processes the output of the LP to reduce the number of rules needed.

Finding the set of tunnels with a given size that carries the most traffic is NP-complete [52]. SWAN uses the following heuristic: first pick at least one tunnel for each DC pair, prefer tunnels that carry more traffic (as per the LP's solution) and repeat as long as more tunnels can be added without violating rule count

constraint m_j at switch j. If M_j is the number of tunnels that switch j can store and $\lambda \in [0, 50\%]$ is the scratch space needed for rule updates (§4.2.2), m_j=$(1-\lambda)M_j$. In practice, we found that $\{m_j\}$ is large enough to ensure at least two tunnels per DC pair (§4.3.3). However, the original allocation of the LP is no longer valid since only a subset of the tunnels are selected due to rule limit constraints. Hence, we re-run the LP, giving it as input only these chosen tunnels. The output of this run has both high utilization and can be implemented in the network.

To further speed-up allocation computation to work with large WANs, SWAN uses two strategies. First, it runs the LP at the granularity of DCs instead of switches. DCs have at least 2 WAN switches, so a DC-level LP has at least 4x fewer variables and constraints (and the complexity of an LP is at least quadratic in this number). To map DC-level allocations to switches, we leverage the symmetry of inter-DC WANs. Each WAN switch in a DC gets equal traffic from inside the DC as border routers use ECMP for outgoing traffic. Similarly, equal traffic arrives from neighboring DCs because switches in a DC have similar fan-out patterns to neighboring DCs. This symmetry allows traffic on each DC-level link (computed by the LP) to be spread equally among the switch-level links between two DCs. However, symmetry may be lost during failures, and can be handled with local topology expansion.

Second, during allocation computation, SWAN aggregates the demands from all services in the same priority class between a pair of DCs. This reduces the number of flows that the LP has to allocate by a factor that equals the number of services, which can run into 100s. Given the per DC-pair allocation, we divide it among individual services in a max-min fair manner.

3.3.3 Handling failures

Gracefully handling failures is an important part of a global resource controller. We outline how SWAN handles failures.

Link and switch failures are detected and communicated to the controller by network agents, in response to which the controller immediately computes new allocations. Some failures can break the symmetry in topology that SWAN leverages for scalable computation of allocation. When computing allocations over an asymmetric topology, the controller expands the topology of impacted

DCs and computes allocations at the switch level directly.

Network agents, service brokers, and the controller have backup instances that take over when the primary fails. For simplicity, the backups do not maintain state but acquire what is needed upon taking over. Network agents query the switches for topology, traffic, and current rules. Service brokers wait for T_h (10 seconds), by which time all hosts would have contacted them. The controller queries the network agents for topology, traffic, and current rule set, and service brokers for current demand. Further, hosts stop sending traffic when they are unable to contact the (primary and secondary) service broker. Service brokers retain their current allocation when they cannot contact the controller. In the period between the primary controller failing and the backup taking over, the network continues to forward traffic as last configured.

3.3.4 Prototype implementation

We have developed a SWAN prototype that implements all the elements described above. The controller, service brokers and hosts, and network agents communicate with each other using RESTful APIs. We implemented network agents using the Floodlight OpenFlow controller [55], which allows SWAN to work with commodity OpenFlow switches. We use the QoS features in Windows Server 2012 to mark DSCP bits in outgoing packets and rate limit traffic using token buckets. We configure priority queues per class in switches. Based on our experiments (§3.5), we set s=10% and λ=10% in our prototype.

3.4 Testbed-based evaluation

We evaluate SWAN on a modest-sized testbed. We examine the network efficiency using today's OpenFlow switches and under TCP dynamics. We will extend our evaluation to the scale of today's inter-DC WANs in §3.5.

3.4.1 Testbed and workload

Our testbed emulates an inter-DC WAN with 5 DCs spread across three continents (Figure 3.5). Each DC has: i) two WAN-facing switches; ii) 5 servers per

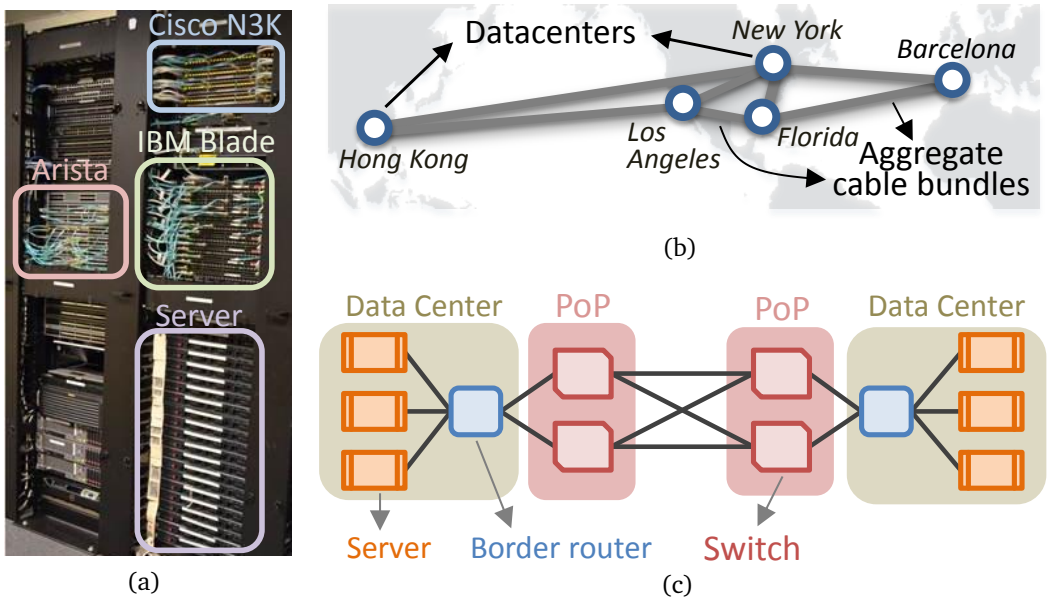


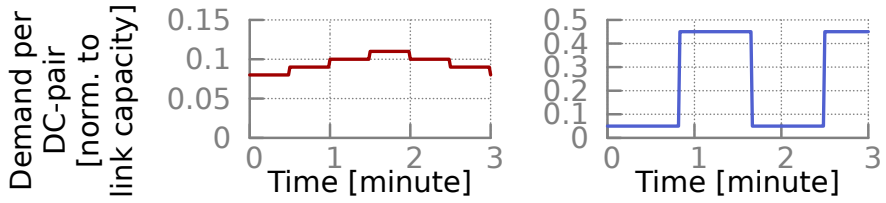
Figure 3.5: Our testbed. (a) Partial view of the equipment. (b) Emulated DC-level topology. (c) Closer look at physical connectivity for a pair of DC.

DC, where each server has a 1G Ethernet NIC and acts as 25 virtual hosts; and iii) an internal router that splits traffic from the hosts over the WAN switches. A logical link between DCs is two physical links between their WAN switches. WAN switches are a mix of Arista 7050Ts and IBM Blade G8264s, and routers are a mix of Cisco N3Ks and Juniper MX960s. The SWAN controller is in New York, and we emulate control message delays based on geographic distances.

In our experiment, every DC pair has a demand in each priority class. The demand of the Background class is infinite, whereas Interactive and Elastic demands vary with a period of 3-minutes as per the patterns shown in Figure 3.6. Each DC pair has a different phase, i.e., their demands are not synchronized. We picked these demands because they have sudden changes in quantity and spatial characteristics to stress SWAN.

The actual traffic per {DC-pair, class} consists of 100s of TCP flows. Our switches do not support unequal splitting, so we insert appropriate rules into the switches to split traffic as needed based on IP headers.

We set T_s and T_c, the service demand and network update frequencies, to one minute, instead of five, to stress-test SWAN's dynamic behavior.



(a) Interactive (b) Elastic

Figure 3.6: Demand patterns for testbed experiments.

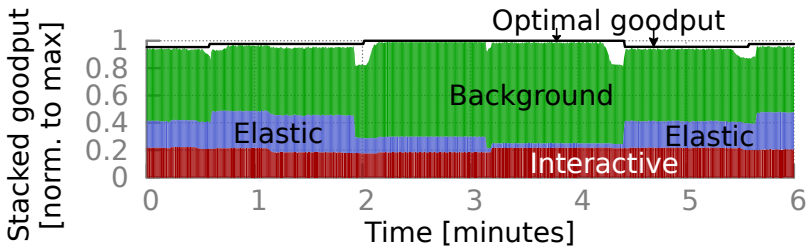


Figure 3.7: SWAN achieves near-optimal throughput.

3.4.2 Experimental results

Efficiency: Figure 3.7 shows that SWAN closely approximates the throughput of an optimal method. For each 1-min interval, this method computes service rates using a multi-class, multi-commodity flow problem that is not constrained by the set of available tunnels or rule count limits. It's prediction of interactive traffic is perfect, it has no overhead due to network updates, and it can modify service rates instantaneously.

Overall, we see that SWAN closely approximates the optimal method. The dips in traffic occur during updates because we ask services whose new allocations are lower to reduce their rates, wait for T_h=10 seconds, and then ask services with higher allocations to increase their rate. The impact of these dips is low in practice when there are more flows and the update frequency is 5 minutes (§3.5.5).

3.5 Performance at scale

To evaluate SWAN at scale, we conduct data-driven simulations with topologies and traffic from two production inter-DC WANs of large cloud service providers (§3.5.1). We show that SWAN can carry 60% more traffic than MPLS

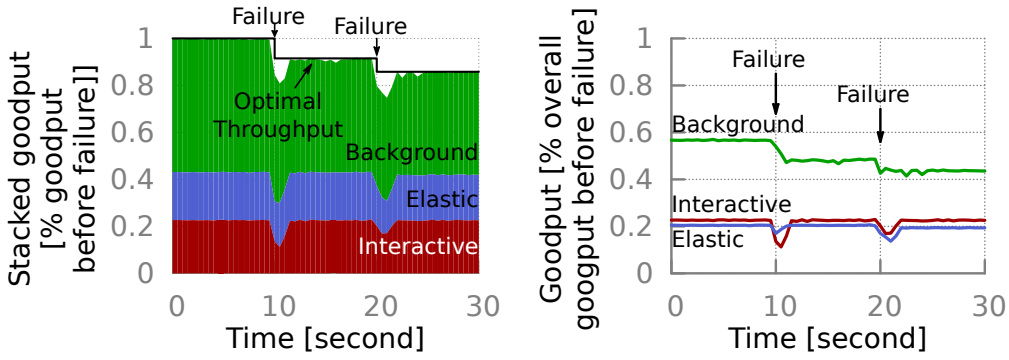


Figure 3.8: SWAN quickly recovers from failures.

TE (§3.5.2) and is also fairer than MPLS TE (§3.5.3).

3.5.1 Datasets and methodology

Inter-DC WAN IDN: A large, well-connected inter-DC WAN with more than 40 DCs. We have accurate topology, capacity, and traffic information for this network. Each DC is connected to 2-16 other DCs, and inter-DC capacities range from tens of Gbps to Tbps. Major DCs have more neighbors and higher capacity connectivity. Each DC has two WAN routers for fault tolerance, and each router connects to both routers in the neighboring DC. We measure flow-level traffic on this network using NetFlow logs collected by routers.

We estimate capacity based on the gravity model [56]. The capacity between two DCs is proportional to the product of their number of neighbors. Reflecting common provisioning practices, we round capacity up to the nearest multiple of 80 Gbps. We obtained qualitatively similar results with three other capacity assignment methods: i) capacity is based on 5-minute peak usage across a week when the traffic is carried over shortest paths using ECMP (we cannot use MPLS-TE as that requires capacity information); ii) capacity between each pair of DCs is 320 Gbps; iii) capacity between a pair of DCs is 320 or 160 Gbps with equal probability.

Through interviews with network operators and service developers, we classify traffic into individual services and map each service to one of the three classes (interactive, elastic, background). We assume that the networks were provisioned such that what we measured was the real demand of services which had not been modulated by capacity limitations.

We conduct experiments using a flow-level simulator that implements a complete version of SWAN. The demand of the services is derived based on the traffic information from a week-long network log. If the full demand of a service is not allocated in an interval, it carries over to the next interval. We place the SWAN controller at a central DC and simulate control plane latency between the controller and entities in other DCs (service brokers, network agents). This latency is based on shortest paths, where the latency of each hop is based on speed of light in fiber and great circle distance.

3.5.2 Network utilization

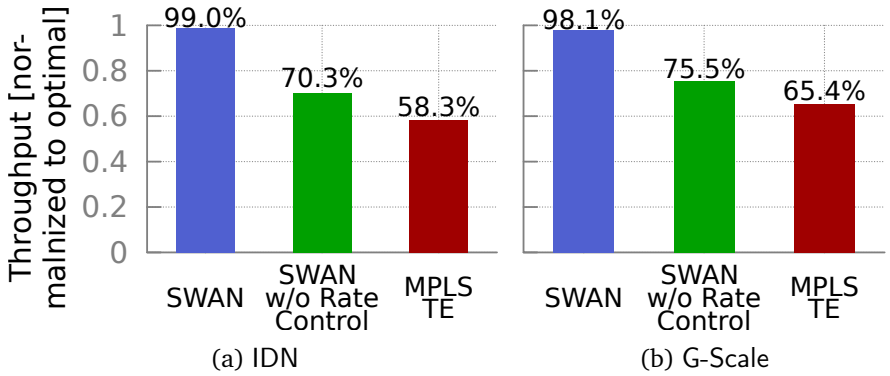
To evaluate how well SWAN utilizes the network, we compare it to an optimal method that can offer 100% utilization. This method computes how much traffic can be carried in each 5-min interval by solving a multi-class, multi-commodity flow problem. It is restricted only by link capacities, not by rule count limits. The changes to service rates are instantaneous, and rate limiting and interactive traffic prediction is perfect.

We also compare SWAN to MPLS TE. Our implementation [43, 44] has the advanced features that IDN uses. Priorities for packets and tunnels protect higher-priority packets and ensure shorter paths for higher-priority services. Per *re-optimization*, CSPF is invoked periodically (5 mins) to search for better path assignments. Per *auto-bandwidth*, tunnel bandwidth is periodically (5 mins) adjusted based on the current traffic demand, estimated by the maximum of the average (across 5-min intervals) demand in the past 15 minutes.

Figure 3.9 shows the traffic that different methods can carry compared to the optimal. To quantify the traffic that a method can carry, we scale service demands by the same factor and use binary search to derive the maximum admissible traffic. We define admissibility as carrying at least 99.9% of service demands.

We see that MPLS TE carries only around 60% of the optimal amount of traffic. SWAN, on the other hand, can carry $> 98\%$ for both WANs. This difference means that SWAN carries over 60% more traffic that MPLS TE, which is a significant gain in the value extracted from the inter-DC WAN.

To decouple gains of SWAN from its two main components—coordination across services and global resource management–we also simulated a variant



(a) IDN (b) G-Scale

Figure 3.9: SWAN carries more traffic than MPLS TE.

of SWAN where the former is absent. Here, instead of getting demand requests from services, we estimate it from their throughput in a manner similar to MPLS TE. We also do not control the rate at which services send. Figure 3.9 shows that this variant of SWAN improves utilization by 10–15% over MPLS TE, i.e., it carries 16–25% more traffic. Even this level of increase in efficiency translates to savings of millions of dollars in the cost of carrying wide-area traffic.

We draw two conclusions from this result. First, both components of SWAN are needed to fully achieve its gains. Second, even in networks where incoming traffic cannot be controlled (e.g., ISP network), worthwhile utilization improvements can be obtained through centralized resource allocation of SWAN.

3.5.3 Fairness

SWAN improves not only efficiency but also fairness. To study fairness, we scale demands such that background traffic is 50% higher than what a mechanism admits; fairness is of interest only when traffic demands cannot be fully met. Further, scaling relative to traffic admitted by a mechanism ensures that oversubscription level is the same. If we used an identical demand matrix for SWAN and MPLS TE, the oversubscription for MPLS TE would be higher as it carries less traffic.

For an exemplary 5-min window, Figure 3.10a shows the throughput that individual flows get relative to their max-min fair share. We focus on background traffic as the higher priority for other traffic means that its demands are often met. We compute max-min fair shares using a computationally-

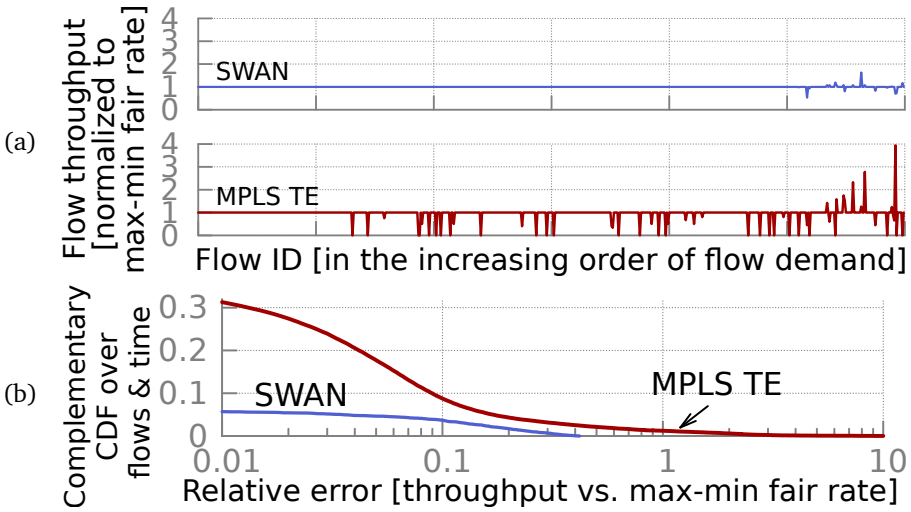


Figure 3.10: SWAN is fairer than MPLS TE.

complex method that is unsuitable for online use [53]. We see that SWAN well approximates max-min fair sharing. In contrast, the greedy, local allocation of MPLS TE is significantly unfair.

Figure 3.10b shows aggregated results. In SWAN, only 4% of the flows deviate over 5% from their fair share. In MPLS TE, 20% of the flows deviate by that much, and the worst-case deviation is much higher. As Figure 3.10a shows, the flows that deviate are not necessarily high- or low-demand, but are spread across the board.

3.5.4 Rule management

A measure of interest for a rule management method is the amount of network capacity it can use given a rule count limit at switches. Figure 4.4 (left) shows this measure for SWAN and an alternative that installs rules for the k-shortest paths between DC-pairs; k is chosen such that the rule count limit is not violated for any switch. We see that, in IDN, k-shortest paths requires 20K rules to fully use the network capacity. As mentioned before, this is beyond what will be offered by next-generation switches. The natural progression towards faster link speeds and larger WANs means that future switches may need even more rules. If switches support 1K rules, k-shortest paths is unable to use 10% of the network capacity. In contrast, SWAN's dynamic tunnels approach enables it to fully use network capacity with an order of magnitude fewer rules.

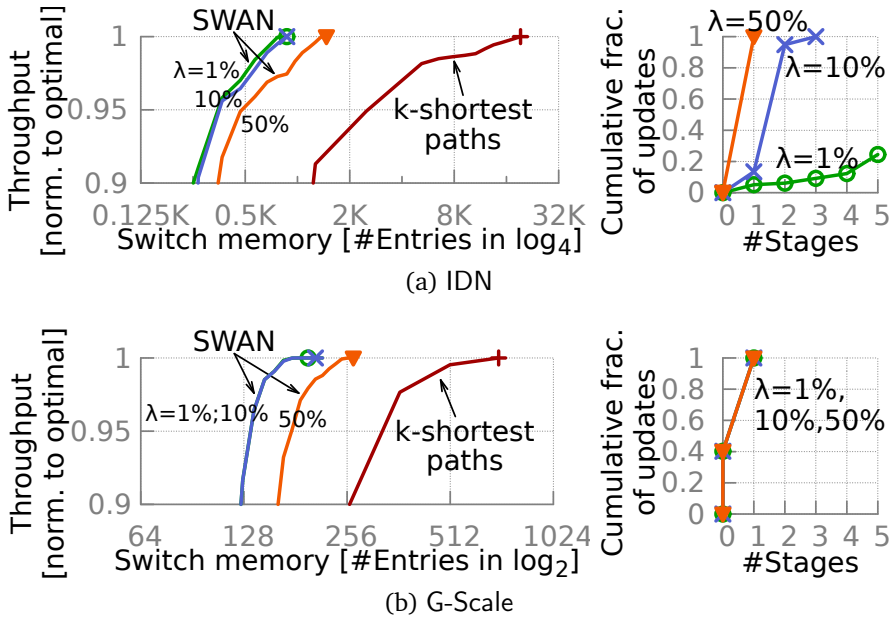


Figure 3.11: SWAN needs fewer rules to fully exploit network capacity (left). The number of stages needed for rule changes is small (right).

This fits within the capabilities of current-generation switches.

3.5.5 Other microbenchmarks

We close our evaluation of SWAN by reporting on some key microbenchmarks.

Update time: Figure 3.12 shows the time to update IDN from the start of a new epoch. Our controller uses a PC with a 2.8GHz CPU and runs unoptimized code. The left graph shows a CDF across all updates. The right graph depicts a timeline of the average time spent in various parts. Most updates finish in 22s; most of this time goes into waiting for service rate limits to take effect, 10s each to wait for services to reduce their rate (t_1 to t_3) and then for those whose rate increases (t_4 to t_5). SWAN computes the congestion-controlled plan in parallel with the first of these. The network's data plane is in flux for only 600 ms on average (t_3 to t_4). This includes communication delay from controller to switches and the time to update rules at switches, multiplied by the number of stages required to bound congestion. If SWAN were used in a network without explicit resource signaling, the average update time would only be this 600 ms.

Traffic carried during updates: During updates, SWAN ensures that the net-

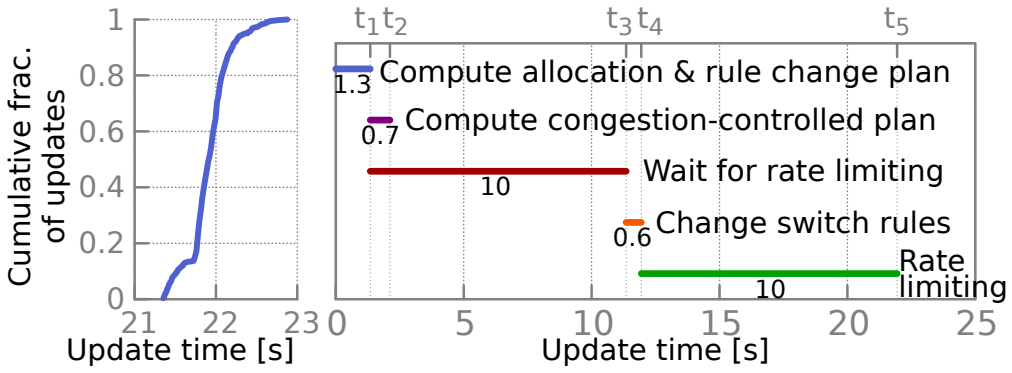
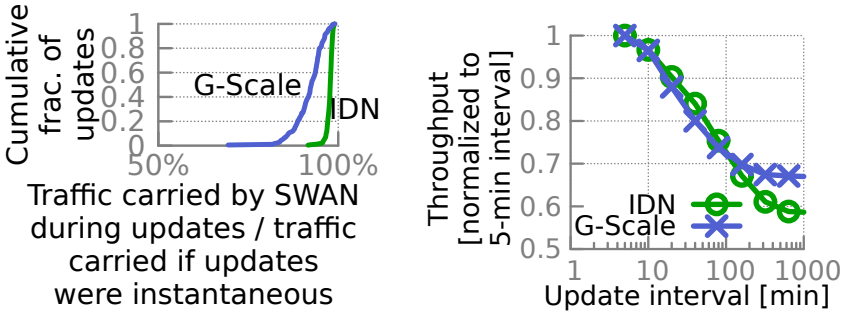


Figure 3.12: Time for network update.



(a) (b)

Figure 3.13: (a) SWAN carries close to optimal traffic even during updates. (b) Frequent updates lead to higher throughput.

work continues to maintain high utilization. That the overall network utilization of SWAN comes close to optimal (§3.5.2) is an evidence of this behavior. More directly, Figure 3.13a shows the %-age of traffic that SWAN carries during updates compared to an optimal method with instantaneous updates. The median value is 96%.

Update frequency: Figure 3.13b shows that frequent updates to the network's data plane lead to higher efficiency. It plots the drop in throughput as the update duration is increased. The service demands still change every 5 minutes but the network data plane updates at the slower rate (x-axis) and the controller allocates as much traffic as the current data plane can carry. We see that an update frequency of 10 (100) minutes reduces throughput by 5% (30%).

Prediction error for interactive traffic: SWAN predicts the amount of interactive traffic in the next epoch. Figure 3.14 shows the error in this prediction. It plots predicted versus actual traffic that traverses a link relative to its capac-

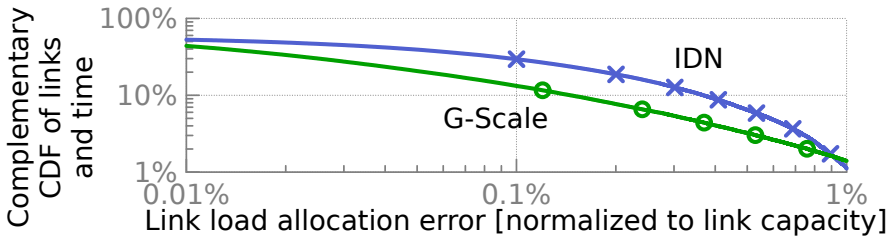


Figure 3.14: Link allocation error due to imperfect demand prediction for interactive traffic.

ity. We see that the error is low, because interactive traffic is stable at these timescales and tends to be a small fraction of link capacity.

3.6 Discussion

This section discusses several issues that, for conciseness, were not mentioned in the main body of the paper.

Non-conforming traffic: Sometimes services may (e.g., due to bugs) send more than what is allocated. SWAN can detect these situations using traffic logs that are collected from switches every 5 minutes. It can then notify the owners of the service and protect other traffic by re-marking the DSCP bits of non-confirming traffic to a class that is even lower than background traffic, so that it's carried only if there is any spare capacity.

Truthful declaration: Services may declare their lower-priority traffic as higher priority or ask for more bandwidth than they can consume. SWAN discourages this behavior through appropriate pricing: services pay more for higher priority and pay for all allocated resources. (Even within a single organization, services pay for the infrastructure resources they consume.)

Richer service-network interface: Our current design has a simple interface between the services and network, based on current bandwidth demand. In future work, we will consider a richer interface such as letting services reserve resources ahead of time and letting them express their needs in terms of total bytes and a deadline by which they must be transmitted. Better knowledge of such needs can further boost efficiency, for instance, by enabling store-and-forward transfers through intermediate DCs [57]. The key challenge here is the design of scalable and fair allocation mechanisms that composes the

diversity of service needs.

3.7 Related work

SWAN builds upon several themes in prior work.

Intra-DC traffic management: Many recent works manage intra-DC traffic to better balance load [28, 46, 58] or share among selfish parties [14, 15, 49]. SWAN is similar to the former in using centralized TE and to the latter in providing fairness. But the *intra*-DC case has constraints and opportunities that do not translate to the WAN. For example, EyeQ [49] assumes that the network has a full bisection bandwidth core and hence only paths to or from the core can be congested; this need not hold for a WAN. Seawall [14] uses TCP-like adaptation to converge to fair share, but high RTTs on the WAN would mean slow convergence. Faircloud [15] identifies strategy-proof sharing mechanisms, i.e., resilient to the choices of individual actors. SWAN uses explicit resource signaling to disallow such greedy actions. Signaling also helps it avoid estimating demands which other centralized TE schemes have to do [28, 46].

WAN TE & SDN: As in SWAN, B4 uses SDNs in the context of inter-DC WANs [42]. Although this parallel work shares a similar high-level architecture, it addresses different challenges. While B4 develops custom switches and mechanisms to integrate existing routing protocols in an SDN environment, SWAN develops mechanisms for congestion-free data plane updates and for effectively using the limited forwarding table capacity of commodity switches.

Optimizing WAN efficiency has rich literature, including tuning the weights of ECMP [59], adapting allocations across pre-established tunnels [60, 61], storing and re-routing bulk data at relay nodes [57], caching at application-layer [62], packet multiplexing [63,64], leveraging reconfigurable optical networks [65]. While such bandwidth efficiency is one of the design goals, SWAN also addresses performance and bandwidth requirements of different traffic classes. In fact, SWAN can help many of these systems by providing available bandwidth information and by offering routes through the WAN that may not be discovered by application-layer overlays.

3.8 Conclusion

By applying SDT to the context of inter-DC WANs, this chapter presents our design, implementation, and evaluation of SWAN, a flexible resource controller that coordinates the sending rates of services and centrally configuring the network data plane. We demonstrate how to use this flexibility to optimize inter-DC WANs' performance, including achieving nearly optimal utilization, service-level prioritization and fairness.

CHAPTER 4

CONGESTION-FREE UPDATE

4.1 Background and motivation

Fine-grained flow control allows the data plane be updated frequently to match traffic demand or network topology changes. However, a key challenge is to implement updates in a way that does not create transient congestion. In this chapter, we study how to meet congestion properties (e.g., bandwidth constraints) during network update and implement a series of network update algorithms in SWAN.

The underlying problem is that the updates are not atomic as they require changes to multiple switches. Even if the before and after states are not congested, congestion can occur during updates if traffic that a link is supposed to carry after the update arrives before the traffic that is supposed to leave has left. The extent and duration of such congestion is worse when the network is busier and has larger RTTs (which lead to greater temporal disparity in the application of updates). Both these conditions hold for our setting, and we find that uncoordinated updates lead to severe congestion and heavy packet loss.

This challenge is fundamental to centralized resource allocation. MPLS-TE's distributed resource allocation can make only a smaller class of "safe" changes; it cannot make coordinated changes that require one flow to move in order to free a link for use by another flow. Further, recent work on atomic updates, to ensure that no packet experiences a mix of old and new forwarding rules [66, 67], does not address our challenge. It does not consider capacity limits and treats each flows independently; congestion can still occur due to uncoordinated flow movements.

We address this challenge by first observing that it is impossible to update the network's data plane without creating congestion if all links are full. SWAN

thus leaves "scratch" capacity s (e.g., 10%) at each link. We prove that this enables a congestion-free plan to update the network in at most $\lceil 1/s \rceil$–1 steps. Each step involves a set of changes to forwarding rules at switches, with the property that there will no congestion independent of the order in which those changes are applied. We then develop an algorithm to find a congestion-free plan with the minimum number of steps. Further, the scratch capacity is not wasted in SWAN. Inter-DC WANs have traffic that is tolerant to small amounts of congestion, e.g., data replication with long deadlines. We extend our basic approach to use all link capacity while guaranteeing bounded-congestion updates for tolerant traffic and congestion-free updates for other traffic.

Another challenge that we face is that fully using network capacity requires many forwarding rules at switches, so that we can exploit many alternative paths through the network; but switch hardware supports a limited number of forwarding rules. Analysis of a production inter-DC WAN shows that the number of rules required to fully use its capacity exceeds the limits of even next generation SDN switches. We address this challenge by dynamically changing, based on traffic demand, the set of paths available in the network. On the same WAN, our technique can fully use network capacity with an order of magnitude fewer rules.

4.2 Design

Our goal is to enable forwarding state updates that are quick and congestion free. We can meet these goals trivially, by simply pausing all data movement on the network during a configuration change. Hence, an added goal is that the network continue to carry significant traffic during updates. Network updates may cause packet re-ordering; in this work, we assume that if needed switch-level mechanisms, such as FLARE [68], or host-level mechanisms, such as reordering robust TCP [69], are in place.

Forwarding state updates are of two types: changing the set of tunnels available in the network, and changing the distribution of traffic across available tunnels. If switch memory had space for every tunnel that may be needed for any traffic demand or topology, then the first type of change is not needed. (In practice, however, rule capacity is scarce and we often have to change tunnels.) In contrast, even if rule capacity was infinite, changes to traffic distri-

bution across tunnels remains a challenge. Recall that atomic changes across switches are hard to achieve and uncoordinated moves can lead to transient congestion. We first describe how we make the second type of change (§4.2.1) and then the first type (§4.2.2).

4.2.1 Updating traffic distribution across tunnels

Given two congestion-free configurations with different traffic distributions, we want to update the network from the first configuration to the second in a *congestion-free* manner. More precisely, let the current network configuration be $C=\{b_{ij} : \forall(i,j)\}$, where b_{ij} is the traffic of flow i over tunnel j. We want to update the network's configuration to $C'=\{b'_{ij} : \forall(i,j)\}$. This update can involve moving many flows, and when an update is applied, the individual switches may apply the changes in any order. Hence, many transient configurations emerge, and in some, a link's load may be much higher than its capacity.

Our goal is to find a sequence of configurations ($C=C_0,\ldots,C_k=C'$) such that no link is overloaded in any configuration. Also, no link should be overloaded when changing from C_i to C_{i+1} regardless of the order in which switches move the individual flows.

In arbitrary cases congestion-free update sequences do not exist (e.g., when all links are full, any first move will congest at least one link). However, if we can engineer the scratch capacity on each link (e.g., s_{Pri} in SWAN; §3.3.2), we show that there exists a congestion-free sequence of updates of length no more than $\lceil 1/s\rceil-1$ (Theorem 2 in Appendix C). The constructive proof of this theorem yields an update sequence with exactly $\lceil 1/s\rceil-1$ steps. But shorter sequences may exist and are desirable because they will lead to faster updates.

We use an LP-based algorithm to find the sequence with the minimal number of steps. Figure 5 shows how to examine whether a feasible sequence of q steps exists. We vary q from 1 to $\lceil 1/s\rceil-1$ in increments of 1. The key part in the LP is the constraint that limits the worst case load on a link during an update to be below link capacity. This load is $\sum_{i,j}\max(b^a_{i,j}, b^{a+1}_{i,j})I_{j,l}$ at step a; it happens when none of the flows that will decrease their contribution have done so, but all flows that will increase their contribution have already done so.

If q is feasible, the LP outputs $C_a=\{b^a_{i,j}\}$, for $a=(1,\ldots,q-1)$, which rep-

Inputs: $\begin{cases} q, & \text{sequence length} \\ b^0_{i,j} = b_{i,j}, & \text{initial configuration} \\ b^q_{i,j} = b'_{i,j}, & \text{final configuration} \\ c_l, & \text{capacity of link } l \\ I_{jl}, & \text{indicates if tunnel } j \text{ using link } l \end{cases}$

Outputs: $\{b^a_{i,j}\}$ $\forall a \in \{1, \ldots q\}$ if feasible

maximize c_{margin} // remaining capacity margin

subject to $\forall i, a : \sum_j b^a_{i,j} = b_i$;

$\forall l, a : c_l \geq \sum_{i,j} \max(b^a_{i,j}, b^{a+1}_{i,j}) \cdot I_{j,l} + c_{margin}$;

$\forall (i, j, a) : b^a_{i,j} \geq 0$;

Algorithm 5: LP to find if a congestion-free update sequence of length q exists.

resent the intermediate configurations that form a congestion-free update sequence.

From congestion-free to bounded-congestion: We showed above that leaving scratch capacity on each link facilitates congestion-free updates. If there exists a class of traffic that is tolerant to moderate congestion (e.g., background traffic in the inter-DC case), then we can get away without leaving the scratch capacity idle with the caveat that transient congestion will only be experienced by traffic in this class. To realize this, first when computing flow allocations (§3.3.2), we use non-zero s_{Pri}=s for interactive and elastic traffic, but set s_{Bg}=0 for background traffic (which is allocated last). Thus, link capacity can be fully used, but no more than $(1-s)$ fraction is used by the higher-priority traffic. Second, instead of congestion-free updates we can bound the extent of congestion, while ensuring that this congestion is only experienced by background traffic. To compute such an update sequence, we replace the per-link capacity constraint in Figure 5 with two constraints, one of which ensures that the worst-case traffic on a link from all classes is no more than $(1+\eta)$ of link capacity ($\eta \in [0, 50\%]$) and the second ensures that the worst-case traffic due to the higher-priority traffic remains below link capacity.

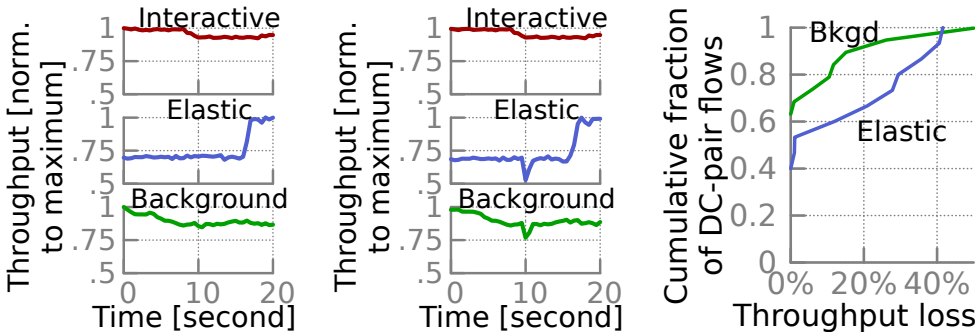
A remaining issue is if bounded-congestion sequences exist and how long they can be. We can prove that if links have slack s with respect to non-background traffic in both C and C', then there exists an update sequence such that i) the non-background traffic on each link is less than its capacity and total traffic is less than $(1+\eta)$ times capacity; and ii) the maximum length of such a sequence is max($\lceil 1/s \rceil$–1, $\lceil 1/\eta \rceil$) (Theorem 3 in Appendix). Based

on this result, we recommend $\eta = \frac{s}{1-s}$.

4.2.2 Updating tunnels

To update the set of tunnels in the network from P to P', we first compute a sequence of tunnel-sets ($P{=}P_0, \ldots, P_k{=}P'$) that each fit within rule limits of switches. Second, for each set, it computes how much traffic from each service can be carried (§3.3.2). Third, it signals services to send at a rate that is minimum across all tunnel-sets. Fourth, after $T_h{=}10$ seconds when services have changed their sending rate, it starts executing the tunnel changes as follows. To go from set P_i to P_{i+1}: i) add tunnels that are in P_{i+1} but not in P_i—the computation of tunnel-sets (described below) guarantees that this will not violate rule count limits; ii) change traffic distribution, using bounded-congestion updates, to what is supported by P_{i+1}, which frees up the tunnels that are in P_i but not in P_{i+1}; iii) delete these tunnels. Finally, SWAN signals to services to start sending at the rate that corresponds to P'.

We compute the interim tunnel-sets as follows. Let $P_{curr}, P_{add}, P_{rem}$ denote the current set of tunnels and those that remain to be added and removed respectively. Initially $P_{curr}{=}P$, $P_{add}{=}P'{-}P$ and $P_{rem}{=}P{-}P'$. Our algorithm proceeds iteratively, picking at each step a subset $P_a \subseteq P_{add}$ to add and a subset $P_r \subseteq P_{rem}$ to flag as being ready to be removed. When selecting from P_{add}, SWAN prefers tunnels that will carry more traffic in the final configuration (P') and those that transit through fewer switches. When selecting from P_{rem}, it prefers rules that carry less traffic in P_{curr} and those that transit through more switches. This biases us towards finding interim tunnel-sets that carry more traffic and use fewer rules. At k^{th} step, we pick the maximal number of tunnels to add (P_a) such that the total added tunnels in the first i steps require at most t^k_{add} rules. The value for t^k_{add} is chosen to ensure $P_{curr} \cup P_a$ fits within memory. Similarly, we pick minimal number of tunnels to remove (P_r) such that the tunnels that remain to be removed require at most t^k_{rem} rules after the removal. The value t^k_{rem} is chosen such that $(P_{curr} \cup P_a) - P_r$ leaves λM_j rule space free at every switch (which is an invariant that holds for P and P' as well; §3.3.2). Then we update the tunnel sets $P_{curr}{=}(P_{curr} \cup P_a){-}P_r$, $P_{add}{-}{=}P_a$, and $P_{rem}{-}{=}P_r$. The process ends when P_{add} and P_{rem} are empty, at which point P_{curr} will be P'.



(a) SWAN (b) One-shot (c) One-shot

Figure 4.1: Updates in SWAN do not cause congestion.

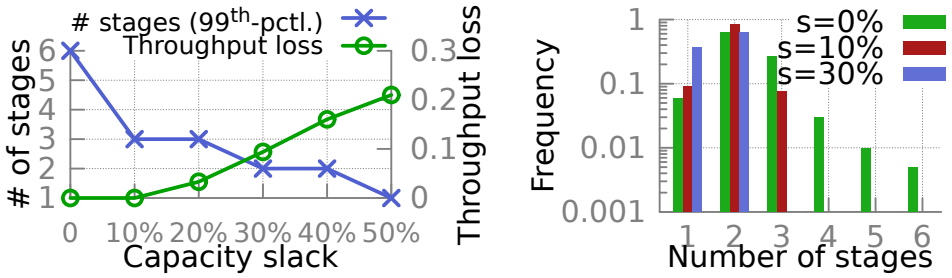
The algorithm above requires at most $\lceil 1/\lambda \rceil - 1$ steps (Theorem 4 in Appendix). At interim steps, some services may get an allocation that is lower than that in P or P'. These services will face a short-term reduction in their rate. The problem of finding interim tunnel-sets in which no service's allocation is lower than the initial and final set, given link capacity constraints, is NP-hard (even a much simpler problems related to rule-limits is NP-hard [52]). In practice, however, we found services rarely experience short-term reductions. Also, since both P and P' contain a common core in which there is at least one common tunnel between each DC-pair (per our tunnel selection algorithm; §3.3.2), basic connectivity is always maintained during transitions, which in practice suffices to carry at least all interactive traffic.

4.3 Performance evaluation

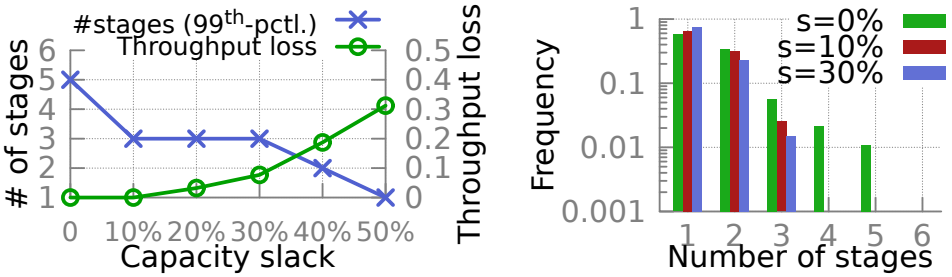
We implement the congestion-free update algorithms in SWAN and evaluate its performance via both testbed and simulation. We show that SWAN enables fast, congestion-controlled network update using bounded switch state (§4.3.3).

4.3.1 Testbed-based evaluation

Congestion-controlled updates: Figure 4.1a zooms in on an example update. A new epoch starts at zero and the throughput of each class is shown relative to its maximal allocation before and after the update. We see that



(a) IDN



(b) G-Scale

Figure 4.2: Number of stages and loss in network throughput as a function of scratch capacity.

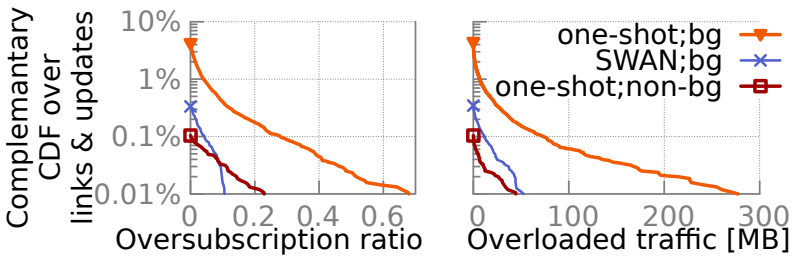
with SWAN there is no adverse impact on the throughput in any class when the forwarding plane update is executed at t=10s.

To contrast, Figure 4.1b shows what happens without congestion-controlled updates. Here, as in SWAN, 10% of scratch capacity is kept with respect to non-background traffic, but all update commands are issued to switches in one step. We see that Elastic and Background classes suffer transient throughput degradation due to congestion induced losses followed by TCP backoffs. Interactive traffic is protected due to priority queuing in this example but that does not hold for updates that move a lot of interactive traffic across paths. During updates, the throughput degradation across all traffic in a class is 20%, but as Figure 4.1c shows, it is as high as 40% for some of the flows.

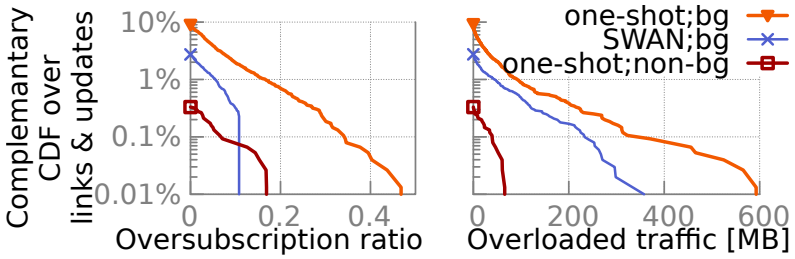
4.3.2 Congestion-controlled updates

We now study congestion-controlled updates via trace-based simulation. We first study the tradeoff regarding the amount of scratch capacity, and then quantify their benefit.

The primary tradeoff when choosing scratch capacity is that higher levels



(a) IDN



(b) G-Scale

Figure 4.3: Link oversubscription during updates.

lead to fewer stages, and thus faster transitions; but they lower the amount of non-background traffic that the network can carry and can lead to wasted capacity if background traffic demand is low. Figure 4.2 show this tradeoff in practice. The left graph plots the maximum number of stages and loss in network throughput as a function of scratch capacity. At the $s = 0$ extreme, throughput loss is zero but more stages are needed to transition safely; infinitely many in the worst case. At the s=50% extreme, only one stage is needed, but the network experiences a throughput loss of 25−36%. The right graph shows the PDF of the number of stages for three values of s. Based on these results, we pick s=10% so that the throughput loss is negligible and updates need only 1-3 stages, much lower than the theoretical worst case of 9.

To show the value of congestion-controlled updates in practice, we consider a method that applies all updates in one shot. This method is identical in every other way Both methods send updates in a stage to the switches in parallel. Each switch applies its updates sequentially and takes 2ms per update [28].

To quantify the benefit from congestion controlled updates, during each reconfiguration, we compute the maximum over-subscription at each link, i.e., load in excess of link capacity as a fraction. Short-lived oversubscription will absorbed into switch queues. Hence, we also compute the maximal buffering

required at each link, i.e., the total excess bytes that arrive during oversubscribed periods. If this number is higher than the size of the physical queue, packets will be dropped. Per priority queuing, we compute oversubscription separately for each traffic class; the computation for non-background traffic ignores background traffic but that for background traffic considers all traffic.

Figure 4.3 shows oversubscription ratios on the left. We see heavy oversubscription with one-shot updates, especially for background traffic. Links can be oversubscribed by up to 60% of their capacity. The right graph plots extra bytes on the links. Today's top-of-line switches, that we use in our testbed, have queue sizes of 9-16 MB. But we see that oversubscription can bring 100s of MB of excess packets and hence, most of these will be dropped. Note that we did not model TCP backoffs which would reduce the load on a link after packet loss starts happening, but regardless, those flows would see significant slowdown. With SWAN, the worst-case oversubscription is 11% ($=\frac{s}{1-s}$) as configured for bounded-congestion updates; this is a better experience for background traffic.

We also see that despite 10% slack, one-shot updates fail to protect even the non-background traffic which is sensitive to loss and delay. Oversubscription can be up to 20%, which can bring over 50 MB of extra bytes during reconfigurations. SWAN fully protects non-background traffic and hence that curve is omitted.

4.3.3 Rule management

A measure of interest for a rule management method is the amount of network capacity it can use given a rule count limit at switches. Figure 4.4 (left) shows this measure for SWAN and an alternative that installs rules for the k-shortest paths between DC-pairs; k is chosen such that the rule count limit is not violated for any switch. We see that, in IDN, k-shortest paths requires 20K rules to fully use the network capacity. As mentioned before, this is beyond what will be offered by next-generation switches. The natural progression towards faster link speeds and larger WANs means that future switches may need even more rules. If switches support 1K rules, k-shortest paths is unable to use 10% of the network capacity. In contrast, SWAN's dynamic tunnels approach enables it to fully use network capacity with an order of magnitude fewer rules.

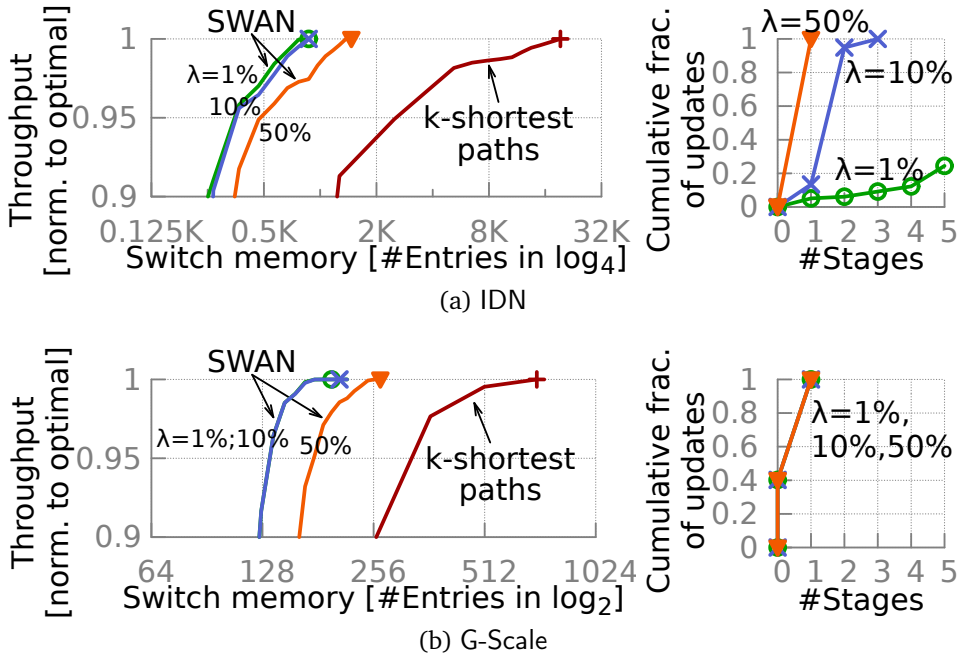


Figure 4.4: SWAN needs fewer rules to fully exploit network capacity (left). The number of stages needed for rule changes is small (right).

This fits within the capabilities of current-generation switches.

Figure 4.4 (right) shows the number of stages needed to dynamically change tunnels. It assumes a limit of 750 OpenFlow rules, which is what the switches in our testbed can support [70]. With 10% slack only two stages are needed 95% of the time; more than three are rarely needed. This nimbleness stems from the efficiency of dynamic tunnels–a small set of rules are needed per interval and some temporal stability in demand matrices– this set changes slowly across adjacent intervals.

4.4 Related work

Some recent work provides guarantees during network updates either on connectivity, or loop-free paths or that a packet will see a consistent set of SDN rules [66, 67, 71, 72]. SWAN offers a stronger guarantee that the network remains uncongested during forwarding rule changes. Vanbever et. al. [72] suggest finding an ordering of updates to individual switches that is guaranteed to be congestion free; however, we see that such ordering may not exist (§4.3.2) and is unlikely to exist when the network operates at high utilization.

4.5 Conclusion

Software-defined flow control can achieve high efficiency by coordinating the sending rates of services and centrally configuring the network data plane. However, network can encounter severe congestion during data plane changes. By leaving a small amount of scratch capacity on the links and switch rule memory, we how how the updates can be implemented quickly and without congestion or disruption.

CHAPTER 5

SCALABILITY AND APPLICATION IMPROVEMENT

5.1 Introduction

SDT lies in the family of *fabric* architectures [21] which use central control to send instructions to edge devices (e.g., end-hosts), allowing the core of the network to provide only basic packet forwarding functions. But the fabric architecture seems infeasible for fine-grained flow-rate control: given the sub-second timescales needed by rate control, centralized scheduling of thousands of servers' flows would raise a serious latency and scalability concern.

The goal of this chapter is to revisit this assumption. How far can we push SDT towards real-time, fine-grained rate control? To answer this question, we present a design and evaluation of an optimized SDT architecture. This design does not support all possible transport policies. Instead, we limit our focus to a nontrival set of transport policies that has been shown to be useful: strict priority scheduling, weighted max-min fairness, or a combination thereof.

Latency is especially an issue at large scales, as the central controller might fail to schedule a large number of flows in real time and could become a serious bottleneck. To make this architecture practical, rate computation has to be fast enough to optimize flow rates in real time at a reasonably large scale. Real-time resource allocation is especially critical as intuitively only flows that last much longer than the control interval of the transport controller would benefit from this architecture.

We present two techniques to push SDT's scalability limit. First, we show a multi-threaded rate allocation algorithm that emulates link-level queueing disciplines. We found such a multi-threaded design greatly reduces computational time by allowing parallel processing of several links. Second, we handle short, transient flows without the central controller. These flows consume little network resource yet typically require timely delivery. Based upon proper

packet header marking (e.g., via DSCP bits) at end-hosts, short flows are initiated with highest queueing priority and do not need to be scheduled by the transport controller until they send more than a certain number of bytes.

The experimental results demonstrate that our prototype implementation can already scale this architecture reasonably well—a single central server can schedule and dynamically re-schedule flow-rates of 95% of bytes in a cluster with several thousand servers within hundreds of milliseconds. Because these 95% of bytes are composed of flows with duration longer than 10 seconds [22], scheduling these flows with a much smaller control interval of hundreds of milliseconds is feasible. This suggests that the centralized rate control architecture is promising for many small and mid-size cloud datacenters or clusters.

We also develop a preliminary implementation of job-level scheduling algorithms. Using the transport policies supported by this design, we demonstrate how to optimize completion times of MapReduce jobs via two transport policies. First, a flow shares network resource with other flows that belongs to the same job with a fair share rate proportional to its flow size. Second, a flow of a smaller job has strictly higher priority than any flow of a larger job. Experimental results show that this design saves average shuffling times by $12 - 20\%$.

5.2 SDT design

We first present an overview of our design (§5.2.1), followed by the transport policies it supports (§5.2.2) and a multi-threaded rate allocation algorithm for scalability (§5.2.3). Finally we present an implementation of this design (§5.2.4).

5.2.1 Overview

At high-level, flows are classified into long and short flow in this optimized version of SDT. We aim to finish short flows faster by assigning a higher queueing priority to short flows, and our central transport controller only rate-limit long flows. This transport controller runs a multi-threaded scheduling algorithm

to dynamically allocate flow-rates based on the service priority predefined by network operator.

Abstractly, the system works as follows.

- A *host agent* is consistently activated on each end-host. When a TCP/UDP flow is established, the host agent keeps track of how many bytes have been acknowledged/sent by the flow. When this number exceeds a threshold of X bytes, the host agent informs the transport controller of the flow.
- The transport controller, which has the current view of the network flows and the routing information, derives the flow sending rates (§5.2.3) based on flow weight and priority pre-defined by the network operator (§5.2.2) and sends the allocation of flow rates back to the host agents.
- The host agent rate-limits its flows based on the rate assigned by the controller.

Letting loose the control of short flows. A threshold X is used to determine the quantity of flows handled by the transport controller for system scalability:

- **Short flow** (has sent $\leq X$ bytes) is neither rate-limited nor known by the transport controller. It it mapped to a higher priority forwarding queue at switches.
- **Long flow** (has sent $> X$ bytes) is rate-limited based on the rate given by the transport controller. To differentiate flows, the host agent marks long flow's DSCP bits such that its outgoing packets will be mapped to a lower priority forwarding queue at switches.

It requires two priority queues at switches, while today's commodity switches typically support $4-8$ priority queues (per port) [13].

5.2.2 Transport policies

Service class. A service class C, consisting of one or multiple flows, is associated with a priority value $p(C)$ and a weight value $w(C)$. The network operator can declare, update, or delete any service classes, and decides how to map flows to service classes. We say an allocation of flow rates respects

service requirements if, and only if, the following properties are satisfied: for any flow i in service classes C_i and flow j in service class C_j:

- If $p(C_i) = p(C_j)$: Flow i and j should share bandwidth in weighted max-min fairness with respect to the weight $w(C_i)$ and $w(C_j)$, respectively.
- If $p(C_i) \neq p(C_j)$: Flow i should have strictly higher (or lower) priority than flow j if $p(C_i) > p(C_j)$ (or $p(C_i) < p(C_j)$).
- The allocation is work conserving, i.e., it is not possible to increase the rate of any flow without decreasing the rate of any other flow.

Transport Policies. By assigning proper priority and weight values to flows, the above definition offers a range of transport policies, for example:

- If we set the same priority and weight to every flow, SDT can approximate flow-level max-min fairness. This emulates an *arbitrarily large* number of processor sharing queues, which otherwise is not supported by commodity switches.
- If we set the same priority but different weights to flows, SDT can approximate weighted max-min fairness. This policy is useful to differentiate flows based on their business priorities (e.g., tenants that pay more should receive a higher total weight for its flows).
- If we set priority differently, SDT can prioritize traffic by emulating an arbitrarily large number of strict priority queues. When the priority is set based on the flow size, this policy has been used to optimize mean flow completion time [9, 10]. When the priority is set based on the flow deadline, this policy has been used to optimize the number of late flows [10].
- When app information is available, SDT can perform application-aware scheduling (e.g., satisfying job deadline [73] and minimal bandwidth guarantee like virtual leased line).

5.2.3 Rate allocation

Our goal is to quickly compute rates for all flows. However, deriving flow rates that respect the service requirements is nontrival. For example, when

computing flow-level max-min fairness, it cannot be done by just iterating through each link and calculating the link-level fair share. One flow's rate can affect others, which in turn affect other flows on other links and so on; essentially, a fluid-level simulation of the entire network is needed.

Our solution is a multi-threaded approach that simulates the fluid-level forwarding behavior on every network link: Based on the input flow rates, a network link individually derives the output flow rates and signals its allocations to downstream neighboring links. By assigning links onto threads in a multi-core processor, this approach naturally yields a better scalability.

Figure 6 shows the pseudocode of SDT's rate allocation algorithm. For each link l, we maintain the following variables:

- l.input$_i$: The input rate of flow i, i.e., an upper bound on the output flow rate.
- l.dirty: The link dirty bit that indicates if we need to recompute the flow rates.
- l.faster$_i$: A binary variable that indicates whether flow i can potentially send faster than the current flow rate. False if the flow is capacity-saturated at some bottleneck link, i.e., the bottleneck link does not have additional capacity to further increase the flow output rate; True if the flow is demand-saturated, i.e., the upper bound on flow rate prevents us from assigning a higher rate to flow i.
- l.token$_i$: A binary variable that indicates whether the link holds flow i's passing token.

Basic operations. At high level, each link l takes l.input$_i$ as the input flow rates and computes the output flow rates for all the flows that traverse through this link. When a link derives a new output flow rate that is different from what it derived previously, it first checks whether itself is the last hop of the flow. If so, the derived output flow rate is the global flow rate, and thus the transport controller sends this new flow rate back to the end-host for rate enforcement. If not, then the link propagates the output rate information by updating the input flow rate on the next-hop link.

Each link updates the allocation by invoking the MPWFQ (Multi-Priority Weighted Fair Queueing) function, which allocates link bandwidth by deriving

weighted fair share separately for flows in an decreasing order of priority (Figure 6). In particular, it first allocates bandwidth to the flows with the highest priority (in weighted max-min fairness based on flow weights), subtracts the allocated bandwidth from the link's remaining capacity, and then repeat the above procedure for the flows with the second highest priority and so on.

Per-link dirty bit. To avoid unnecessary computation, each link should call MPWFQ only when its inputs have changed (e.g., the input rate of a flow has decreased because of an upstream bottleneck). To this end, we use a dirty bit per link to indicate whether the inputs were modified. In particular, we mark a link as dirty when we change its input flow rate, and we recompute the flow rates only when the link is dirty.

Because a dirty bit could have multiple writing and reading threads, one way to avoid the concurrency issue is to use mutex to protect each link. However, the overhead of placing mutexes could be large as it might temporarily block several threads and slow down the subsequent operations. Fortunately, we found that mutexes can be avoided in this case: When a link is dirty, the working thread cleans the dirty bit *before* it calls MPWFQ. When we need to update the input flow rate on the next-hop link, we mark the next-hop link as dirty *after* we finish the update of input flow rate. link. Because flipping a bit is an atomic operation, it can be shown that the above implementation will not generate any false negative case (i.e., a dirty link gets stuck in the clean state).

Source feedback. However, the procedure described above does not guarantee work-conservation: When a flow gets bottlenecked on a link, its upstream links are not aware of this situation and can overallocate this flow too much resource, which otherwise can be allocated to other competing flows. To fix this, the last-hop link sends feedback to the first-hop link of a flow whenever the output rate has changed. Let R_{out} be the output rate on the last-hop link and R_{in} be the input rate on the first-hop link:

- If $l.\text{faster}_i = false$ on the last-hop link l, we set R_{in} to R_{out}. Intuition: There exists a bottleneck link which asks the flow to reduce its rate to R_{out}. We set the input rate to R_{out} such that the upstream links will not overallocate capacity to this flow.

- If $l.\text{faster}_i = true$ on the last-hop link l, we set R_{in} to b_i. Intuition: The flow can potentially send faster if we had increased the input rate. For

work-conserving, we increase the input rate R_{in} to its maximal value: the flow demand b_i.

Feedback rate control with tokens. For stability, we assign a per-flow token to its first-hop link. When a link holds the token of a flow, it passes the token to the next-hop link after it recomputes the flow rates. The source feedback is allowed only when the last-hop link holds the token. This token-based constraint avoids over-reaction as it ensures that feedback will occur only when the previous feedback has being taken effect on every link the flow traversed. Empirically we found this token-based constraint helps the system converges to the target flow rates quickly. We leave the proof of stability as future work.

5.2.4 Prototype implementation

We have implemented the transport controller in C++11 using Boost Asio library [74] and Libcurl APIs [75] for control message communication, the host agent in Python with Twisted library [76]. For routing, we use Floodlight OpenFlow controller [55]. The control messages are all sent over TCP. We implement per-flow rate limiting using tc rate-limiter and implement packet marking using iptables in the Linux kernel. The end-hosts use iPerf3 [77] to generate TCP flows.

5.3 Evaluation

In this section, we first present our evaluation setting (§5.3.1), followed by a series of experiments aiming to answer the following two questions: *Scalability* (§5.3.2)—what's the largest scale of network that SDT can scheduling at flow-level granularity in real-time? *Flexibility* (§5.3.3)—does our prototype support useful transport policies that help improve network performance, including application-level improvement?

5.3.1 Evaluation setting

Testbed. To study scalability and flexibility, we have built a testbed (Figure 5.1) to evaluate SDT's scalability and flexibility. We have four servers

Inputs:
$F_{Pri,l}$: The set of active flows that traversed link l with priority Pri
b_i: The bandwidth demands of flow i
w_i: The weight of flow i
c_l: The capacity of link l
Outputs:
r_i: rate allocation of flow i

Func: Rate Allocation Algorithm
$\forall l : l$.dirty $\leftarrow$ true;
foreach (l, i) **do**
  l.input$_i \leftarrow b_i$; l.faster$_i \leftarrow$ false;
  l.token$_i \leftarrow$ (Is link l the first-hop link of flow i?);
end
parallel.while *each link* l **do**
  if $l.dirty = false$ **then continue**;
  l.dirty = false; $R_l \leftarrow$ MPWFQ(l);
  foreach $r_{l,i} \in R_l$ **do** MPA($l, i, r_{l,i}$);
endwhile

// Emulating multi-priority weight fair queueing at link l.
Func: MPWFQ(l):
$c_l^{remain} \leftarrow c_l$; $R_l \leftarrow \emptyset$;
for $Pri = Highest, \ldots, Lowest$ **do**
  Compute local rate allocations $\{r_{l,i} : i \in F_{Pri,l}\}$, the weighted max-min fair share rate for allocating capacity c_l^{remain} to each flow $i \in F_{Pri,l}$ with weight w_i, subject to the rate upper bound l.input$_i$;
  $R_l \leftarrow R_l + \{r_{l,i}\}$; $c_l^{remain} \leftarrow c_l^{remain} - \sum_{i \in F_{Pri,l}} r_{l,i}$;
end
return R_l; // Return the rate allocations at link l

// Message passing algorithm for flow i at link l with a new output rate $r_{l,i}$.
Func: MPA($l, i, r_{l,i}$):
if *link* l *is flow* i*'s last hop* **then**
  $r_i \leftarrow r_{l,i}$; Let l^* be flow i's first-hop link;
  if $l.token_i = true$ **then**
    if $l.faster = false$ *and* $l^*.input \neq r_i$ **then**
      l^*.input$_i \leftarrow r_i$; l.token$_i \leftarrow$ false;
      l^*.token$_i \leftarrow$ true; l^*.dirty $\leftarrow$ true;
    end
    if $l.faster = true$ *and* $l^*.input \neq b_i$ **then**
      l^*.input$_i \leftarrow b_i$; l.token$_i \leftarrow$ false;
      l^*.token$_i \leftarrow$ true; l^*.dirty $\leftarrow$ true;
    end
  end
end
else
  $d \leftarrow$ false; Let l' be flow i's next-hop link;
  if $l'.input_i \neq r_{l,i}$ **then**
    l'.input$_i = r_{l,i}$; $d \leftarrow$ true;
  end
  if $l.token_i = true$ **then**
    l'.token$_i \leftarrow$ true; l.token$_i \leftarrow$ false; $d \leftarrow$ true;
  end
  $f \leftarrow l$.faster && (is $r_{l,i}$ demand-saturated?);
  if $f \neq l'.faster$ **then** l'.faster $\leftarrow faster$; $d \leftarrow$ true;
  if $d = true$ **then** l'.dirty $\leftarrow$ true;
end

Algorithm 6: Computing rate allocations.

installed in the bottom of the testbed rack. Each server installed VMware vSphere Hypervisor (ESXi) to run $28 - 30$ Linux VMs. Each VM is equipped with a physical 1-Gbps Ethernet NIC. At the top of the rack we place 13 Pronto

Figure 5.1: Our testbed.

3290 switches running OpenFlow v1.0. Although we interconnect these devices randomly, our default topology is a sub-network which forms a two-level tree. This tree topology is rooted by a core virtual switch connecting to 20 top-of-rack virtual switches, each of which has further connected to $4-6$ servers. A virtual switch consists of a set of ports on a physical switch. The transport controller runs on Dell PC (PowerEdge T620) with an Intel Xeon E5-2630L (2 processor sockets, 6 cores per socket, and a total of 24 logical processors).

Transport controller. The transport controller pre-configured virtual topology routing path in the following steps: (1) Run Floodlight to detect network physical topology using Link Layer Topology Discovery (LLTD) protocol. (2) Form a virtual topology on top of physical topology using a brute-force search fashion subject to a constraint where each virtual link is bijectively mapping to a physical cable. (3) Floyd-Warshall algorithm is used to compute inter-VM routing paths. (4) We derived the static forwarding rules based on the routing using ingress port, destination IP address, and DSCP bits. (5) The static rules

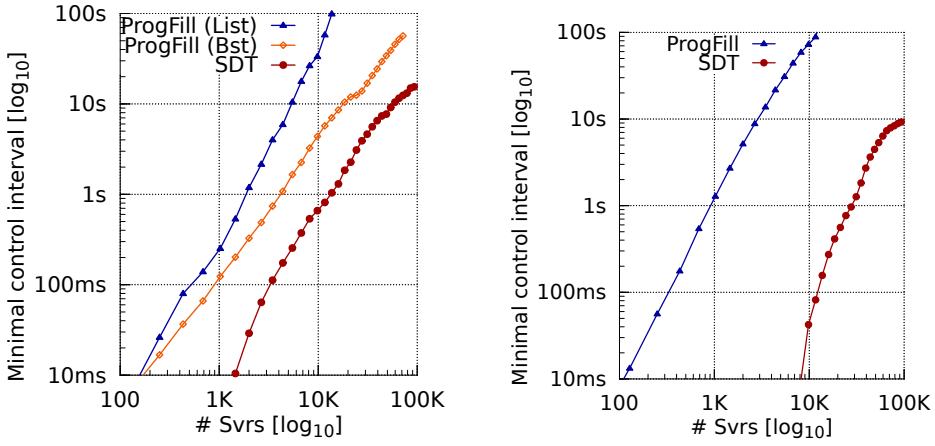


Figure 5.2: SDT runs much faster than progress filling algorithm. SDT handles $> 95\%$ of the bytes in a datacenter/cluster of several thousands servers with a control interval of a few hundreds of milliseconds. Two tested transport policies: (a) fair-sharing, (b) prioritization.

are pushed to switches via Floodlight's Static Flow Pusher API.

We have implemented the following schemes.

SDT: We have developed a full version of the optimized SDT in our testbed, including the multi-threaded rate allocation algorithm, per-link dirty bit, and token-based source feedback. Based on our results, we empirically set the flow size detection threshold to 80 MBytes.

ProgFill: We implemented an extended version of the progressive filling algorithm. We first classify flows based on its priority, and then we run the progress filling algorithm per priority class. In each priority group, we iteratively find bottleneck links by incrementally increasing flow rates proportional to their fair-share weight and remove the bottleneck links from the network. The bottleneck link is defined as the link which first gets saturated their available capacity when increasing flow rates. Our basic implementation maintains a list of active, unsorted network links and updates each link to find the current bottleneck link in each iteration. We also implemented an optimized solution which stores links using a binary search tree to reduce the time complexity.

5.3.2 Scalability

To study the scalability of the transport controller, we conduct a microbenchmark experiment to virtually create a network, feed the network and traffic information to the transport controller and measure how long the controller takes to converge to the target rates. We emulate fat-tree topology with varying sizes [34], and the generated flow inter-arrival times and flow durations are drawn from the distributions measured in a production query-processing cluster [22]. We compute the *minimal control interval*, denoted by T, using a binary search procedure, while satisfying the following constraints:

- Given the current set of active network flows, the controller needs to derive the sending rates for every flow controlled by SDT within an average computational time of T seconds.
- The controller does not schedule short flows and only handles longer flows whose duration is larger than $10T$; We look for a control interval that is an order of magnitude smaller than the flow duration in order to ensure that the additional control latency imposed by SDT's control feedback loop does not significantly delay the shorter flows that otherwise could complete faster.

Figure 5.2 shows that, using a commodity PC, our implementation of transport controller can scale to a network with several thousands of servers while operating at a control interval of only a few hundreds of milliseconds. Given the common long-tail traffic characteristics in datacenters and clusters [2,22], a central transport controller with a control interval of several hundreds of milliseconds, which is fast enough to handle flows that lasts more than 10 seconds, can schedule more than 98% of the total network bytes [2,22]. On the other hand, as more than half the network bytes are in flows that appear ≤ 25 seconds in DC traffic [22], it is critical to ensure that the control interval is not larger than a few seconds to make the central rate control more useful. With this constraint, the current implementation can support close to ten thousands of servers with a control interval of one second.

SDT scales better when handling prioritization over fair sharing because our algorithm converges faster for flows with distinct strict priorities. The intuition of this observation is as follows. When SDT computes fair sharing schedule,

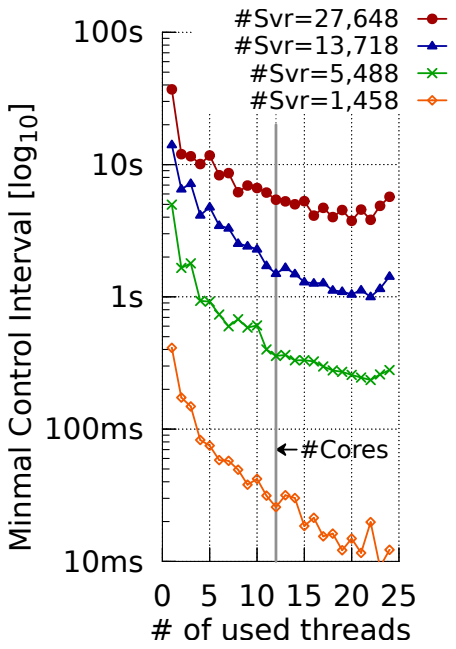


Figure 5.3: SDT scales well with the number of used threads. The scale-up of using multiple threads is near linear when the number of used threads is no more than the number of cores.

flows typically converges after triggering source feedback many times. However, in flow prioritization, flows that have highest priority at the moment will guarantee to converge to stable state without source feedback.

We also compare SDT with progress filling algorithm. Our implementation of progressive filling algorithm includes an optimization of bottleneck search using binary search tree. Note that this optimization only applied to the fair sharing but not flow prioritization case.

In Figure 5.2, we observe that SDT schedules fair sharing flows 4-10x faster than our optimized progress filling algorithm. When prioritizing network flows, SDT runs $\sim$ 22x faster than the baseline approach. These observations demonstrate that SDT can scale to schedule much larger networks given the required control interval.

We also found a near linear scale-up when using multiple threads (Figure 5.3). However, this improvement per thread becomes smaller when the number of used threads goes larger than the number of physical cores in our testbed server. We see an overall scale-up ratio of 12-21x when using all 24 threads.

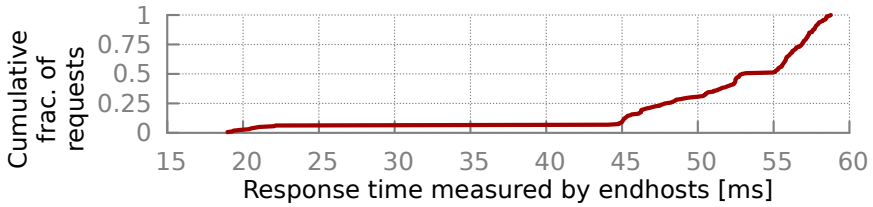


Figure 5.4: Most flows wait < 58 ms to receive their final rate allocation. The experiment runs across 1,600 randomly destined flows that started simultaneously.

The experiments above consider only the computational overhead occurred inside the transport controller and did not include other sources of latency such as the latency induced by exchanging control messages between the transport controller and end-hosts. Here, we extend our experiment to study the control response latency experienced by flows at *end-host side*: The end-to-end response time is measured from the transmission of flow arrival information to the receipt of the final rate allocation.

Figure 5.4 shows the empirical distribution of the end-to-end response time across 1,600 randomly destined flows that started simultaneously in our testbed. We see a short-tailed distribution; a majority of the flows experienced an end-to-end response time of $45 - 58$ ms. Note that flows started without being rate-limited and may receive interim rate allocations before the receipt of its final rate. Therefore, during a control interval, the flow sending rate is merely a gradual shift away from their ideal rate based on the configured transport policies towards conventional TCP-based bandwidth sharing.

5.3.3 Flexibility

This section first presents a case study to demonstrate how SDT helps finish cluster application faster, followed by three scenarios to show SDT can dynamically adapt to handle a variety of cases.

Improving MapReduce Shuffling Time. When applying SDT to MapReduce jobs, the goal is to complete *job* faster. In MapReduce workload, the delay of completion time contributed by networking is solely determined by the duration of the entire shuffle phase (i.e., moving data from mappers to reducers). Therefore, we aim to minimize the completion time of the *last* flow, which decides the duration of the shuffle phase. To this end, we set flow's weight to

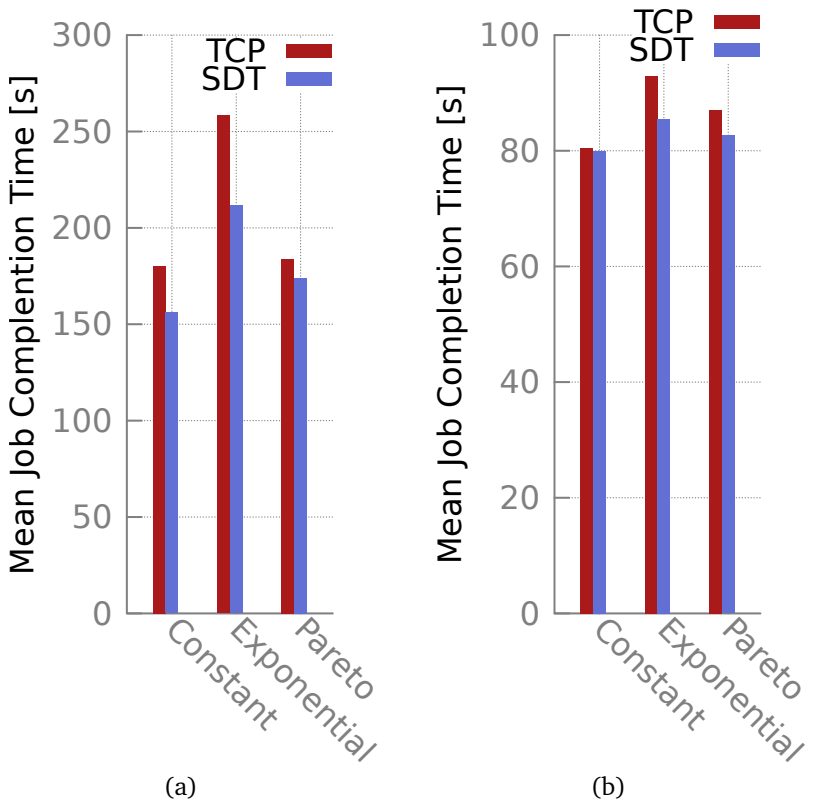


Figure 5.5: SDT completes shuffle phase of MapReduce jobs faster. For each job, we allocate 9 mappers and 9 reducers randomly from our VM pool subject to the constraint that each VM serves either a mapper or reducer and a VM is only used by a job at any time. Each job has a total of 81 shuffling flows. (a) 5 concurrent jobs; (b) jobs are executed in serial.

its size, and we set flow's priority to the inverse of the job size. This setting emulates the following transport policy configuration.

- For any two competing flows of a job, they share the network bandwidth in a weighted max-min fairness fashion with respect to their size. This setting helps longer flow to finish faster and helps mitigate the bad cases where the whole job is bottlenecked by several long flows.

- For any two flows belonging to different jobs, the flow in the smaller job has strictly higher priority than that in the larger job. This setting helps complete short jobs first and serves as a good heuristic to minimize average job completion time.

Figure 5.5 demonstrates that SDT help complete shuffle phase of MapRe-

duce jobs faster. For applications that generate similar flow sizes, e.g., sorting, we found SDT saves $\sim$ 12% mean job completion time (see the constant case in Figure 5.5a). We use exponential and Pareto distributions to approximate the flow sizes for other types of applications. We found SDT saves $6 - 20\%$ mean job completion time.

The shuffling phases of flows with non-constant sizes are generally longer than that of constant-sized flows because the long flow usually becomes the straggler and delays the completion time of shuffle phase. We see SDT can greatly mitigate such non-uniform network usage when the long flow's tail is smaller (e.g., Exponential distribution in Figure 5.5). However, the improvement becomes smaller under heavy-tailed distribution like Pareto because even a single elephant flow could create severe network bottleneck. Transport rate control does not solve this case even when allowing long flows to send with maximal possible rate.

We also compare with the case where jobs are executed in serial (Figure 5.5b). As the improvement decreases significantly as compared with the previous scenario, this experiment helps us break down the contribution of our each policy: Performance gains mainly comes from inter-job prioritization; intra-job weighted fair-share scheduling has limited contribution to minimize mean job completion times, especially for the case of constant flow sizes.

Scenario #1: Flow prioritization. Figure 5.6 shows a scenario where multiple flows shared a common bottleneck link. We demonstrate that SDT, being able to prioritizing flows, finishes flows much faster than traditional TCP-based rate control; we see a savings of 35% in mean flow completion time. The overall goodput in SDT is 4.3% lower than that in TCP primarily due to the fact that updating the rate parameter takes $\sim$ 100 ms in our unoptimized token bucket implementation.

Scenario #2: Fairness. Figure 5.7 shows a scenario where a flow that has multiple bottlenecks receives only $\sim$ 25% of its max-min fair share rate. In comparison, SDT achieves max-min fairness when assigning the same priority and weight to every flow.

Scenario #3: Controlling long flows only. Figure 5.8 demonstrates that SDT can avoid scheduling short flows for better scalability. By mapping SDT-controlled flows to a lower priority forwarding queue at switches, long flows will automatically back off and short flows can finish quickly without interven-

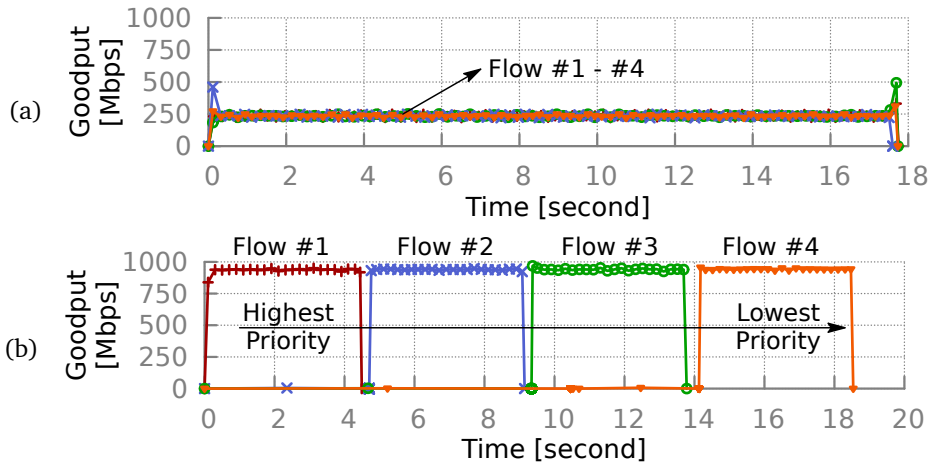


Figure 5.6: SDT can optimize flow completion time by prioritizing flows. Consider a scenario where four flows (with a size of 500 MB) shared a single bottleneck link: (a) TCP with a mean flow completion time of 17.7 seconds; (b) SDT with a mean flow completion time of 11.49 seconds.

tion by the transport controller, as evident in Figure 5.8.

5.4 Related work

Fine-grained SDT is close to OpenTCP [78], a SDN-based control layer that dynamically adjusts TCP parameters and variants based on the network traffic measurements. SDT differs from OpenTCP in at least two ways. First, SDT offers greater flexibility than TCP-based congestion control, because the ability to control flow sending rates explicitly allows network operators to implement a wider range of transport policies with SDT. For example, when fairness is desirable, we show a scenario where flows can quickly converge to max-min fairness in SDT but not in TCP (Scenario #2 in §3.5). Second, OpenTCP did not show that explicit centralized rate control is feasible. With $\sim$ 4,000 hosts, OpenTCP centrally controlled TCP parameters and variants with a control interval of 60 seconds. In SDT, we show that the more challenging task of explicit network-wide rate control is feasible at a single server with a control interval that is 100x faster than OpenTCP at the same scale.

Many works on rate control of network traffic are distributed solutions at either flow level (e.g., D^3 [13], D2TCP [11], PDQ [10], DeTail [12], pFabric [9]) or server level (e.g., NetShare [79], Seawall [14], FairCloud [15],

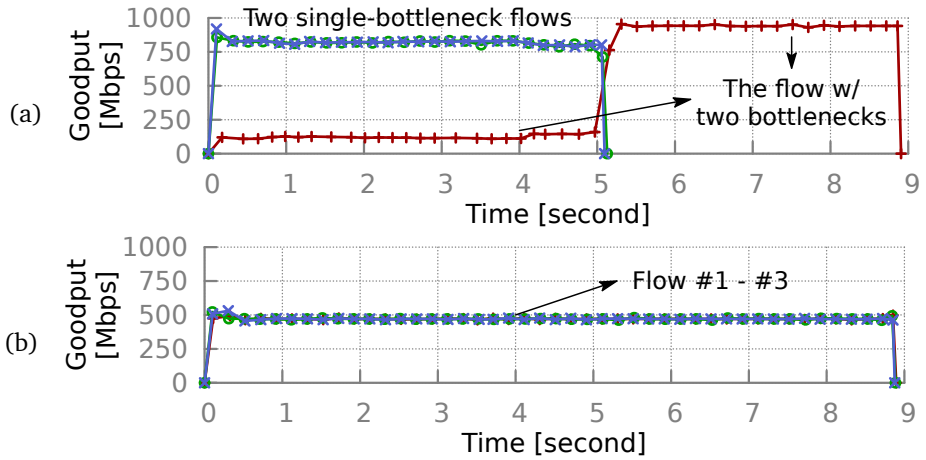


Figure 5.7: SDT can achieve fairness when needed. Consider a flow (colored in red) with two bottlenecks, at each of which it competes the network bandwidth with another flow. (a) TCP fails to provide fairness by assigning a lower rate to the multiple-bottleneck flow; (b) SDT ensures each flow receives its max-min fair share rate.

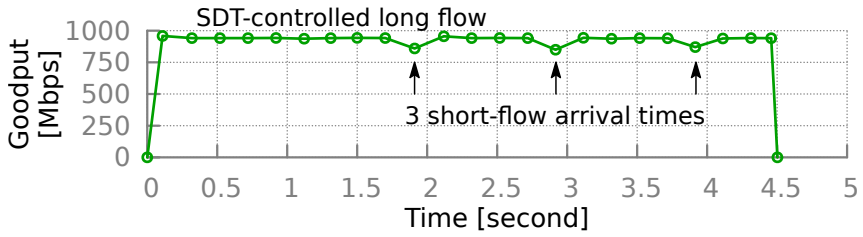


Figure 5.8: SDT achieves scalability by letting loose of the control of short flows. Consider a scenario where three short flows of 2 MB interrupts a background long flow. The short flows (time series are omitted in the figure) finish within < 20 ms with an average goodput of ~ 900 Mbps.

EyeQ [49], ElasticSwitch [16]). But they lack the programmability of a centralized controller and are often difficult to deploy. More recently, B4 [42], Varys [80] and PASE [81] leverage a centralized software rate controller that achieves greater deployability for inter-datacenter networks. However, most of these works allocate network bandwidth at a much coarser level (e.g., B4 operates at per {datacenter-pair, QoS class}-tuple). It is unclear how to extend these centralized solutions to handle explicit rate control at flow level in real-time.

5.5 Conclusion

We present an optimization design of SDT to improve its scalability. We found our implementation can already rate-control most of the traffic at flow-level in many small and mid-sized cluster and datacenters. We also demonstrated its potential to improve application-level performance.

We expect service requirements will continue to emerge, prompting network operators to continue to support new transport policies. Our current prototype, of course, does not support all the possible policies; instead, it serves as a nontrival example and provides several transport policies that have been shown to be very useful to network operators. Moreover, many policies might be easily implementable on top of the current prototype. For example, an earliest-deadline-first scheduling can be implemented in our design by (i) assigning flow priorities inversely proportional to their time to deadline, and (ii) terminating/de-prioritizing flows that failed to meet their deadlines.

CHAPTER 6

FUTURE WORK AND CONCLUSION

In this chapter, we discuss some possible future directions and draw a conclusion for this dissertation.

6.1 Future work

Handling erroneous input: we showed that network performance can be optimized by leveraging application information (e.g, traffic demand and deadline). However, the application information may be erroneous, and thus the centralized transport controller needs to be robust against erroneous information. The research challenges here include: (1) designing pricing-based algorithms to provide proper incentives for truthful demand declaration, and (2) developing proper mechanisms to penalize misbehaving applications and provide worst-case performance guarantees for the rest of the applications.

Network service abstractions: Today's networks lack high-level abstractions for application developers to specify their requirements. Many service-level agreements (e.g., latency targets, bandwidth requirements, service availability) are not visible to applications, and application developers often need to work with network operators in order to ensure network provides desirable properties for the running applications. We suggest to explore the design space of network service abstraction set that provides application developers high-level semantics of network transport features.

Optimizing forwarding rules: Today's switch hardware supports a limited number of forwarding rules, which stems from the amount of fast, expensive memory in switches. However, our centralized transport control relies on the ability to frequently update forwarding state in the network, but doing so requires temporarily rule space to maintain the desirable network properties during the transitions. Compressing rulesets (e.g., using coarse-grained

wild-card rules) helps reduce the needed rules, but complicates the ability to update the network. In particular, it would be interesting to understand the tradeoffs between two important factors: (1) the minimal number of rules needed to implement certain transport policies, and (2) the temporarily rule space required to update the network while satisfying these policies during the updates.

Failures. Handling controller failures is an integral part of a centralized transport architecture. Because of the large scale of cloud network infrastructure, we anticipate that failures will occur inevitably. Naively, one can always run a hot-backup instance of controller to take over the control when the primary instance fails. During the recovery period after controller failure, end-hosts should continue sending data and simply fall back to operate without rate limiting. SDT can be seen as a flow-rate optimization towards specific service requirements, and therefore a controller failure affects performance but not correctness.

Evaluation on more cases: Chapter 3 shows that our system could achieve high utilization with bounded update congestion under two inter-datacenter networks; Chapter 4 shows that our system improves application-level performance under fat-tree networks. For future work, we should like to test the performance under a border set of input cases, including network topologies, traffic sending patterns, flow size distributions. A better understanding of how sensitive our methods to network changes will not only serve as a good guideline for parameter selection but also shed some light for future improvement. For example, heterogeneous capacity slack seems to make sense in some cases.

6.2 Conclusion

This dissertation advocated SDT, a software-defined transport architecture and studied how to design and implement methods for optimizing network resource in real time for cloud infrastructure. We first showed that flexible, fine-grained flow-rate control enables a wide range of transport policies and can help optimize network performance to satisfy service requirements. We then applied SDT to inter-datacenter WAN and developed a system to show SDT significantly boosts network utilization while satisfying service priority. We proposed network update algorithms and show how, by leaving a small

amount of scratch capacity on the links and switch memory, the network updates can be implemented quickly and provably congestion-free. We proposed a fast, multi-threaded resource allocation to push the scalability limit further—a single central server can schedule and dynamically re-schedule flow-rates of most of the network bytes in a cluster with several thousand servers within hundreds of milliseconds. Finally, we developed a preliminary implementation of job-level scheduler on top of SDT and showed that SDT saves mean shuffling times of MapReduce jobs by 12-20%.

We hope the practical methods developed in this dissertation close the gap between software-defined networking and production cloud network infrastructure.

APPENDIX A

PROOFS FOR CHAPTER 2

A.1 Deadlock-freedom

Deadlock is a situation where two or more competing flows are paused and are each waiting for the other to finish, and therefore neither ever does. We verify PDQ has no deadlock by showing that *hold and wait*, a necessary condition of deadlock, is false. Hold and wait is a situation that a flow is accepted by some intermediate switches, while paused by other switches along the routing path. In PDQ, a flow is accepted *only* after *every* switch along the path accepts the flow. Moreover, if a PDQ flow is paused, the switch who pauses this flow will update the pauseby field in the scheduling header ($\mathscr{P}_H$) to its ID. Hence, after some time goes by, this information will reach all the other switches along the path, as even the paused flows would send probes periodically. Whenever a switch notices that a flow is paused by another switch, it will not consider accepting this flow. Thus, a paused flow will not be accepted by *any* switch along the path.

A.2 Bounding the convergence time

Assumptions: Without loss of generality, we assume there is no packet loss. Similarly, we assume flows will not be paused due to the use of flow dampening. Because PDQ flows periodically send probes, the properties we discuss in this section will hold with additional latency when the above assumptions are violated. For simplicity, we also assume the link rate C is equal to the maximal sending rate $\mathscr{R}_S^{\max}$ (i.e., $\mathscr{R}_S^{\text{sch}} = 0$ or C). Thus, each link accepts only one flow at a time.

Definitions: We say a flow is *competing* with another flow if and only if they

share at least one common link. Moreover, we say a flow F_1 is a *precedential* flow of flow F_2 if and only if they are competing with each other and flow F_1 is more critical than flow F_2. We say a flow F is a *driver* if and only if (i) flow F is more critical than any other competing flow, or (ii) all the competing flows of flow F that are more critical than flow F are non-drivers.

Lemma 1. When all the precedential flows of a flow F are paused (or it has none) and workload is stable (no new flows arrive and no sending flow finishes), flow F will be accepted in at most one RTT. If any precedential flow of a flow F is accepted, flow F will be paused in at most one RTT.

Proof. This property follows directly from the PDQ flow controller algorithm. □

Lemma 2. PDQ will converge to the equilibrium in $P_{\max}+1$ RTTs for stable workloads, where $P_{\max}$ is the maximal number of precedential flows of any flow. Given a collection of active flows, the equilibrium is defined as a state where all the drivers are accepted while the remaining flows are paused.

Proof. We show that when the workload is stable (no new flows arrive and no sending flow finishes), a flow will be accepted if it is a driver and will otherwise be paused in $P+1$ RTTs, where P is the number of its precedential flows. We prove this will hold for any flow F that is the m-th critical flow in the network by induction on m. When $m=1$, flow F is a driver by definition. Thus, it will be accepted in one RTT according to Lemma 1. When $m=n+1$, there exist n flows $F_1 \cdots F_n$ that are more critical than flow F. Without loss of generality, out of these n flows, we assume there are $n' \leq n$ precedential flows (as they are competing with flow F). Suppose that flow F is a driver. Then, all these n' flows are non-drivers by definition. By the induction hypothesis, these n' competing flows will all be paused in $P'+1$ RTTs, where P' is the maximal possible number of precedential flows of these n' flows. As each of these n' flows will have at most $n-1$ precedential flows, we have $P' \leq n-1$. After these flows are paused, it takes at most an RTT for switches to accept flow F according to Lemma 1, and therefore the flow F will be accepted in $(P'+1)+1 \leq n+1$ RTTs. Suppose now that flow F is not a driver. According to the definition of driver, among those n' competing flows, there exists >0 drivers. Similarly, among these drivers, the maximal number of precedential flows of each driver is at most $n-1$. By the induction hypothesis, these drivers

will be accepted in at most n RTTs, and after this, flow F will be paused in one RTT according to Lemma 1. □

APPENDIX B

PROOFS FOR CHAPTER 3

Property 1. *Let r_i be the max-min fair rate of flow i,and b_i be the rate allocated to flow i by SWAN's Approx Max-Min Fairness algorithm (Figure 4), also let $U < \min r_i$. Then $b_i \in \left[\frac{r_i}{\alpha}, \alpha r_i\right]$.*

Proof Sketch: Observe that the linear program (MCF) per se maximizes overall throughput $\sum b_i$ with a bias towards carrying more of the traffic on paths that have less weight (e.g., shorter paths); ϵ is a small constant. However, Approx Max-Min Fairness invokes MCF in T steps with the constraint that at step k, flows are allocated rates in the range $\left[\alpha^{k-1}U, \alpha^k U\right]$ but no more than their demand. A flow's allocation is *frozen* at step k when it is allocated its full demand d_i at that step or it receives a rate smaller than $\alpha^k U$ due to capacity constraints.

The algorithm's allocation proceeds in steps. Our proof proceeds in epochs that consist of one or more steps. One of three things can happen at each step – first, no link is newly saturated at that step; second, some links are saturated but every link has at least one flow using it that is not capacity saturated, i.e, the flow has other links and paths that it can send more traffic on; third, some links are saturated and all flows on those links are capacity saturated. Note that, by definition, at steps of the first type if a flow is frozen it has to be because its demand is saturated $b_i = d_i$. Because otherwise, the throughput maximization objective will cause that flow to get the maximum possible rate at those steps. The same holds at steps of the second type, because capacity can be freed up on a saturated link by moving some of the traffic belonging to its unsaturated flows off that link. Only at steps of the third type could there be flows frozen because they are limited by capacity. We say that the ongoing epoch ends and a new one begins after each step of the third type.

We will prove that the maximal unfairness for flows that are frozen in each epoch is bounded. Note, that at the end of an epoch, every flow using one of the newly saturated links in this epoch will also be frozen by definition.

Hence, these flows and links can be removed from the topology since nothing changes for either at subsequent steps.

To prove this, first note that every flow that is frozen because its demand has been met or it is capacity saturated has rate equaling its max-min fair rate; since other flows that remain unfrozen at the time this flow freezes receive at least this much rate. Second, we divide flows that are frozen due to capacity limits into groups such that two flows will be in the same group if they send non-zero traffic on at least one common link. Within a group, the rate could be allocated unfairly. However, the total rate allocated to these flows remains the same; allocating less reduces overall throughput and allocating more is not possible since the group is capacity constrained. Further since a group of flows simultaneously freezes at the same stage (of type three), it means that the ratio of the lowest flow rate to the largest flow rate in the group is α and the fair rate of the flow falls somewhere in between.

□

APPENDIX C

PROOFS FOR CHAPTER 4

Property 2. *If all links in the network have a relative slack s, in both the initial flow C and and the final flow C′, then there exists a congestion-free sequence of updates of length no more than* $\lceil 1/s \rceil - 1$.

Proof. We prove this constructively. Let $b^0_{i,j} = b_{i,j}$ denote the allocated bandwidth in the initial configuration, and let $b^q_{i,j} = b'_{i,j}$ denote the allocated bandwidth in the final configuration, after q steps. The superscript $0, \ldots, q$ refer to the update stages. In each stage, we increase the allocated bandwidth by $(b^q_{i,j} - b^0_{i,j})/q$. This algorithm ensures:

- After $q = \lceil 1/s \rceil - 1$ stages, we reach the allocated bandwidth in the final configuration as $b^0_{i,j} + q \cdot (b^q_{i,j} - b^0_{i,j})/q = b^q_{i,j}$.

- After the k^{th} stage, we need to show the network configuration is still valid, i.e., $\sum_j b^k_{i,j} = b_i$, the total bandwidth of flow i. Because each $b^k_{i,j}$ is a linear combination of $b^0_{i,j}$ and $b^q_{i,j}$, we have $\min(b^0_{i,j}, b^q_{i,j}) \leq b^s_{i,j} \leq \max(b^0_{i,j}, b^q_{i,j})$. By adding up all possible tunnels, we have $\min(\sum_j b^0_{i,j}, \sum_j b^q_{i,j}) \leq \sum_j b^k_{i,j} \leq \max(\sum_j b^0_{i,j}, \sum_j b^q_{i,j})$. Because the original and target configurations are valid, we must have $b_i = \sum_j b^0_{i,j} = \sum_j b^q_{i,j}$. Therefore, $\sum_j b^k_{i,j} = b_i$. Likewise, the flow is valid at every node in every step.

- After the k^{th} stage, every link has slack s. Let w^k_l denote the link l's load after k^{th} stage, we have $w^k_l = \sum_{i,j} b^k_{i,j} \cdot I_{j,l}$. Because the network has a slack s in both original and target configurations, we have $w^0_l \leq (1-s)c_l$ and $w^q_l \leq (1-s)c_l$. Because w^k_l is a linear combination of w^0_l and w^q_l, we have $w^k_l \leq (1-s)c_l$.

- During each stage, we need to show any transient configuration will not overload any link. Let $\Delta^k_{i,j}$ denote the increase of flow i's rate at tunnel j during the k^{th} stage, we have $\Delta^k_{i,j} = b^{k+1}_{i,j} - b^k_{i,j}$. Consider the

worst update sequence for link l where all the tunnels with increased rate ($\Delta_{i,j}^k > 0$) are already updated, while the tunnels with decreased rate are not updated. In the beginning of the stage, link l has a residual capacity sC_l. Now all the paths with increase rate are updated, the increase of load is $\sum_{i,j;\Delta_{i,j}^k>0} I_{j,l} \cdot (b_{i,j}^q - b_{i,j}^0)/q \leq \sum_{i,j} I_{j,l} \cdot b_{i,j}^q/q \leq s \cdot \sum_{i,j} I_{j,l} \cdot b_{i,j}^q/(1-s) \leq sC_l$.

□

Property 3. *If non-background traffic has slack s in both C and C', then there exists an update sequence such that (i) non-background traffic does not have congestive loss, (ii) the maximal congestion for background traffic at any link is bounded by ηC_l, and (iii) the maximum length of the update sequence is $max(\lceil 1/s \rceil - 1, \lceil 1/\eta \rceil)$.*

Proof. We reuse the same algorithm as given in the proof for Theorem 2 but the maximum length of the update sequence is set to $q = \max(\lceil 1/s \rceil - 1, \lceil 1/\eta \rceil)$ here. Because the non-background traffic goes at higher priority on the data plane, and given $q \geq \lceil 1/s \rceil - 1$, the algorithm ensures that non-background traffic does not have congestive loss by Theorem 2. Thus, we only need to show background traffic can be dropped no more than ηC_l at any link l. Let $\Delta_{i,j}^k$ denote the increase of flow i's rate at tunnel j during the k^{th} stage, we have $\Delta_{i,j}^k = b_{i,j}^{k+1} - b_{i,j}^k$. Consider the *worst* update sequence for link l where all the tunnels with increased rate ($\Delta_{i,j}^k > 0$) are already updated, while the tunnels with decreased rate are not updated. The maximal congestion is at most the total increase of load G:

$$\begin{aligned} G &\leq \sum_{i,j;\Delta_{i,j}^k>0} I_{j,l} \cdot (b_{i,j}^q - b_{i,j}^0)/q \\ &\leq \sum_{i,j} I_{j,l} \cdot b_{i,j}^q/q \\ &\leq \eta \cdot \sum_{i,j} I_{j,l} \cdot b_{i,j}^q \\ &\leq \eta C_l. \end{aligned}$$

□

Property 4. *If any switch j has a memory slack $\lambda \cdot M_j$ in P and P', then the rule change algorithm requires at most $z = \lceil 1/\lambda \rceil - 1$ steps and satisfies the memory constraint.*

Proof. Recall that at i^{th} step SWAN picks tunnels to add such that the total added tunnels in the first i steps require at most t_i^{add} rules. Also, SWAN picks tunnels to remove such that the tunnels that remain to be removed require at most t_i^{rem} rules after the removal. Let r_j, r'_j and r^*_j be the number of rules used by $P - P'$, $P' - P$ and $P \cap P'$, respectively. We define the per-step rule addition limit, denote by s_j, to be $M_j - \max(r_j, r'_j) + r^*_j$. Then we set $t_i^{add} = i \cdot s_j$ and $t_i^{rem} = \max(0, M_j - r^*_j - (1+i)s_j)$.

We first show the algorithm will terminate after $z = \lceil 1/\lambda \rceil - 1$ steps. This holds if $t_z^{add} = z \cdot s_j \geq r'_j + r^*_j$ because we will have enough capacity to select entire P' at any switch j. Also, because P and P' provide a slack of λM_j, we have $r_j \leq (1-\lambda)M_j - r^*_j$ and $r'_j \leq (1-\lambda)M_j - r^*_j$. Therefore,

$$\begin{aligned} z \cdot s_j &= (\lceil 1/\lambda \rceil - 1) \cdot (M_j - \max(r_j, r'_j) - r^*_j) \\ &\geq (1/\lambda - 1) \cdot (\lambda M_j) \\ &= (1-\lambda)M_j \\ &\geq r'_j + r^*_j \end{aligned}$$

Next, we show the algorithm satisfies the memory constraint at any switch. At i^{th} step, the highest memory load happens when new tunnels have already added to switches but old tunnels have not deleted. The total added tunnels in the first i steps contribute at most $t_i^{add} = i \cdot s_j$ rules, and the tunnels that remain to be removed require at most $t_{i-1}^{rem} = \max(0, M_j - r^*_j - i \cdot s_j)$ rules. Also, each switch stores a static set of tunnels $P \cap P'$ that requires at most r^*_j rules. Adding these up, the maximal memory load is at most $(i \cdot s_j) + \max(0, M_j - r^*_j - i \cdot s_j) + (r^*_j) \leq M_j$. □

REFERENCES

[1] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network," in *SIGCOMM*, 2009.

[2] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *IMC*, 2010.

[3] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *OSDI*, 2004.

[4] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *EuroSys*, 2007.

[5] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "SCOPE: easy and efficient parallel processing of massive data sets," in *VLDB*, 2008.

[6] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "FlumeJava: easy, efficient data-parallel pipelines," in *PLDI*, 2010.

[7] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD*, 2010.

[8] "Apache Giraph," http://giraph.apache.org/.

[9] M. Alizadeh, S. Yang, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "Deconstructing datacenter packet transport," in *HotNets*, 2012.

[10] C.-Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," in *ACM SIGCOMM*, 2012.

[11] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter TCP (D2TCP)," in *SIGCOMM*, 2012.

[12] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz, "DeTail: Reducing the flow completion time tail in datacenter networks," in *SIGCOMM*, 2012.

[13] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron, "Better never than late: Meeting deadlines in datacenter networks," in *SIGCOMM*, 2011.

[14] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha, "Sharing the data center network," in *NSDI*, 2011.

[15] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "FairCloud: sharing the network in cloud computing," in *SIGCOMM*, 2012.

[16] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos, "ElasticSwitch: practical work-conserving bandwidth guarantees for cloud computing," in *SIGCOMM*, 2013.

[17] W.-C. Feng, D. D. Kandlur, D. Saha, and K. G. Shin, "Understanding and improving tcp performance over networks with minimum rate guarantees," *IEEE/ACM Trans. Netw.*, 1999.

[18] A. Arefin, R. Rivas, R. Tabassum, and K. Nahrstedt, "Opensession: Sdn-based cross-layer multi-stream management protocol for 3d teleimmersion," in *ICNP*, 2013.

[19] D. Katabi, M. Handley, and C. Rohrs, "Congestion control for high bandwidth-delay product networks," in *SIGCOMM*, 2002.

[20] N. Dukkipati and N. McKeown, "Why flow-completion time is the right metric for congestion control," *SIGCOMM Comput. Commun. Rev.*, 2006.

[21] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian, "Fabric: a retrospective on evolving SDN," in *HotSDN*, 2012.

[22] S. Kandula et al., "The nature of data center traffic: measurements & analysis," in *IMC*, 2009.

[23] J. Brutlag, "Speed matters for Google web search," 2009.

[24] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *SIGCOMM*, 2010.

[25] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "ICTCP: incast congestion control for TCP in data center networks," in *CoNext*, 2010.

[26] N. Bansal and M. Harchol-Balter, "Analysis of SRPT scheduling: Investigating unfairness," in *SIGMETRICS*, 2001.

[27] N. Bansal and M. Harchol-Balter, "End-to-end statistical delay service under GPS and EDF scheduling: A comparison study," in *INFOCOM*, 2001.

[28] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *SIGCOMM*, 2011.

[29] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[30] N. Dukkipati, Y. Ganjali, and R. Zhang-Shen, "Typical versus worst case design in networking," in *HotNets*, 2005.

[31] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller, "Safe and effective fine-grained TCP retransmissions for datacenter communication," in *SIGCOMM*, 2009.

[32] M. L. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, 2nd ed. Springer, 2002.

[33] "Bro network security monitor," http://www.bro-ids.org.

[34] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *SIGCOMM*, 2008.

[35] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "BCube: a high performance, server-centric network architecture for modular data centers," in *SIGCOMM*, 2009.

[36] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: Networking data centers randomly," in *USENIX NSDI*, 2012.

[37] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley, "Design, implementation and evaluation of congestion control for multipath TCP," in *NSDI*, 2011.

[38] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath tcp," in *SIGCOMM*, 2011.

[39] M. Roughan, S. Sen, O. Spatscheck, and N. Duffield, "Class-of-service mapping for QoS: A statistical signature-based approach to IP traffic classification," in *IMC*, 2004.

[40] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: low-overhead datacenter traffic management using end-host-based elephant detection," in *INFOCOM*, 2011.

[41] C. Labovitz, S. Iekel-Johnson, D. McPherson, J. Oberheide, and F. Jahanian, "Internet inter-domain traffic," *SIGCOMM Comput. Commun. Rev.*, 2010.

[42] S. Jain et al., "B4: Experience with a globally-deployed software defined WAN," in *SIGCOMM*, 2013.

[43] D. Awduche, J. Malcolm, J. Agogbua, M. O'Dell, and J. McManus, "Requirements for traffic engineering over MPLS," RFC 2702, 1999.

[44] M. Meyer and J. Vasseur, "MPLS traffic engineering soft preemption," RFC 5712, Internet Engineering Task Force, 2010.

[45] A. Pathak, M. Zhang, Y. C. Hu, R. Mahajan, and D. Maltz, "Latency inflation with MPLS-based traffic engineering," in *IMC*, 2011.

[46] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *NSDI*, 2010.

[47] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *SIGCOMM*, 2011.

[48] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine grained traffic engineering for data centers," in *CoNEXT*, 2011.

[49] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg, "EyeQ: practical network performance isolation at the edge," in *NSDI*, 2013.

[50] K.-S. Lui, W. C. Lee, and K. Nahrstedt, "Star: A transparent spanning tree bridge protocol with alternate routing," *SIGCOMM Comput. Commun. Rev.*, 2002.

[51] Y. Chen, S. Jain, V. K. Adhikari, Z.-L. Zhang, and K. Xu, "A first look at inter-data center traffic characteristics via Yahoo! datasets," in *INFOCOM*, 2011.

[52] T. Hartman, A. Hassidim, H. Kaplan, D. Raz, and M. Segalov, "How to split a flow?" in *INFOCOM*, 2012.

[53] D. Nace, N.-L. Doan, E. Gourdin, and B. Liau, "Computing optimal max-min fair resource allocation for elastic flows," *IEEE/ACM Trans. Netw.*, 2006.

[54] E. Danna, S. Mandal, and A. Singh, "A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering," in *INFOCOM*, 2012.

[55] "Project Floodlight," http://www.projectfloodlight.org/.

[56] M. Roughan, A. Greenberg, C. Kalmanek, M. Rumsewicz, J. Yates, and Y. Zhang, "Experience in measuring backbone traffic variability: Models, metrics, measurements and meaning," in *Internet Measurement Workshop*, 2002.

[57] N. Laoutaris, M. Sirivianos, X. Yang, and P. Rodriguez, "Inter-datacenter bulk transfers with NetStitcher," in *SIGCOMM*, 2011.

[58] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with Orchestra," in *SIGCOMM*, 2011.

[59] B. Fortz, J. Rexford, and M. Thorup, "Traffic engineering with traditional IP routing protocols," *IEEE Comm. Mag.*, 2002.

[60] A. Elwalid, C. Jin, S. Low, and I. Widjaja, "MATE: MPLS adaptive traffic engineering," in *INFOCOM*, 2001.

[61] S. Kandula, D. Katabi, B. Davie, and A. Charny, "Walking the tightrope: Responsive yet stable traffic engineering," in *SIGCOMM*, 2005.

[62] S. Traverso, K. Huguenin, I. Trestian, V. Erramilli, N. Laoutaris, and K. Papagiannaki, "Tailgate: handling long-tail content with a little help from friends," in *WWW*, 2012.

[63] D. Ferrari and D. Verma, "A scheme for real-time channel establishment in wide-area networks," *IEEE Journal on Selected Areas in Communications*, 1990.

[64] D. Ferrari, "A new admission control method for real-time communication in an internetwork," *Technical report*, 1995.

[65] A. Mahimkar, A. Chiu, R. Doverspike, M. D. Feuer, P. Magill, E. Mavrogiorgis, J. Pastor, S. L. Woodward, and J. Yates, "Bandwidth on demand for inter-data center communication," in *HotNets*, 2011.

[66] R. McGeer, "A safe, efficient update protocol for OpenFlow networks," in *HotSDN*, 2012.

[67] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *SIGCOMM*, 2012.

[68] S. Kandula, D. Katabi, S. Sinha, and A. Berger, "Dynamic load balancing without packet reordering," *SIGCOMM CCR*, 2007.

[69] M. Zhang, B. Karp, S. Floyd, and L. Peterson, "RR-TCP: A reordering-robust TCP with DSACK," in *ICNP*, 2003.

[70] R. Recio, "Software defined networking – Disruptive technologies," http://www.cs.arizona.edu/~bzhang/CCW2012/slides/recio.pdf, 2012.

[71] N. Kushman, S. Kandula, D. Katabi, and B. M. Maggs, "R-BGP: Staying connected in a connected world," in *NSDI*, 2007.

[72] L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, and O. Bonaventure, "Seamless network-wide IGP migrations," in *SIGCOMM*, 2011.

[73] M. Chowdhury and I. Stoica, "Coflow: a networking abstraction for cluster applications," in *HotNets*, 2012.

[74] "Boost C++ library," http://www.boost.org/.

[75] "Libcurl: The multiprotocol file transfer library," http://curl.haxx.se/libcurl/.

[76] "Twisted library," http://twistedmatrix.com/.

[77] "Iperf3: TCP and UDP bandwidth performance measurement tool," http://code.google.com/p/iperf/.

[78] M. Ghobadi, S. H. Yeganeh, and Y. Ganjali, "Rethinking end-to-end congestion control in software-defined networks," in *HotNets*, 2012.

[79] T. Lam, S. Radhakrishnan, A. Vahdat, and G. Varghese, "NetShare: virtualizing data center networks across services," in *UCSD Technical Report*, 2010.

[80] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient Coflow Scheduling with Varys," in *SIGCOMM*, 2014.

[81] A. Munir, G. Baig, S. M. Irteza, I. A. Qazi, A. X. Liu, and F. R. Dogar, "Friends, not foes: Synthesizing existing transport strategies for data center networks," in *SIGCOMM*, 2014.